Fractal Architectures & Complex Adaptive Systems

"The Gnomon Project"

final technical report

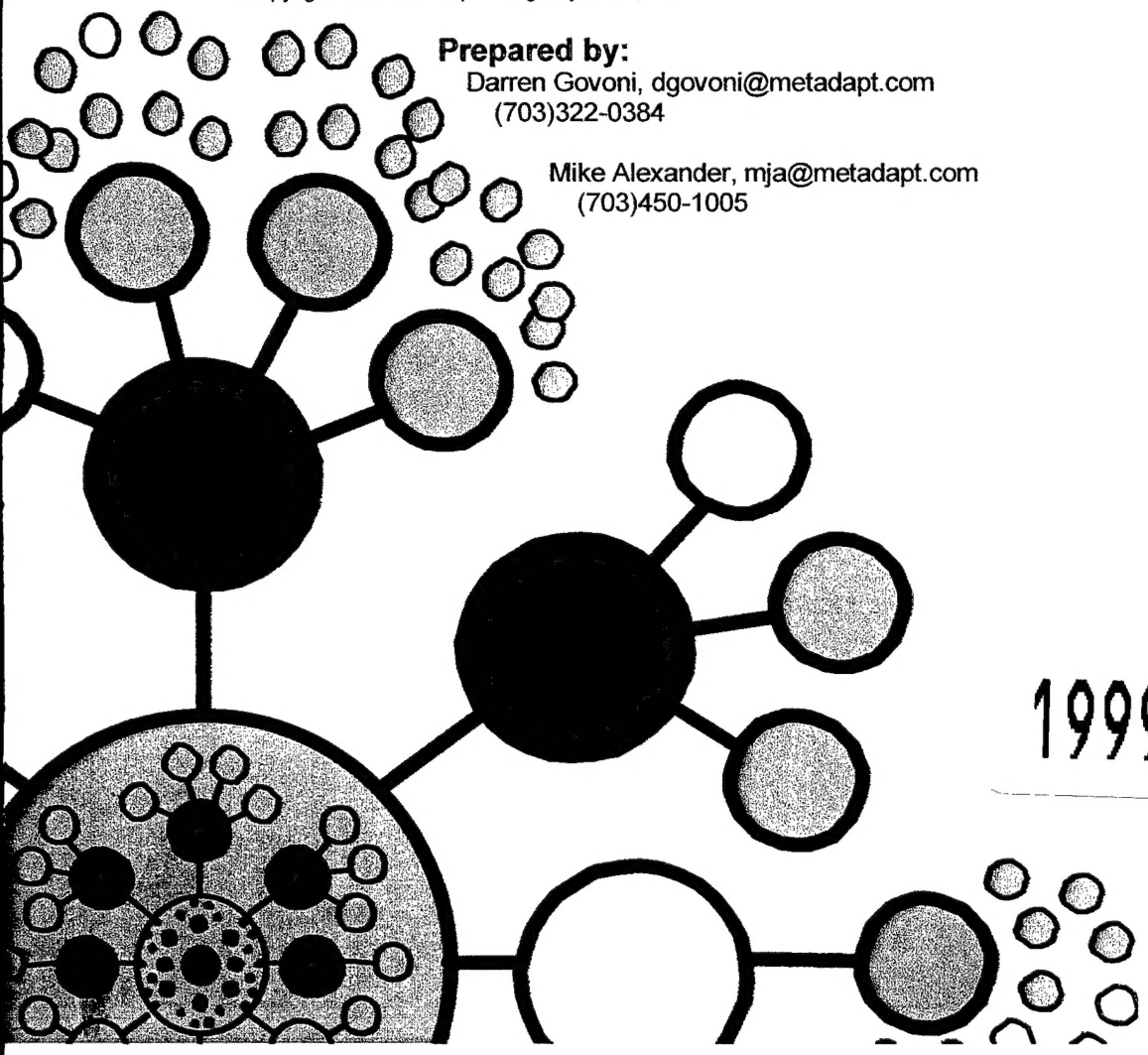
Period:
May 1, 1999 - Oct 31, 1999

Contract Number:
DAAH01-99-C-R118

Contractor Title:
Metadapt Design Systems, Inc.
11706 Tumbrel Ct, Fairfax, VA 22030

Security Classification:
UNCLASSIFIED

Distribution Statement A:
Approved for dissemination and duplication.



Prepared by:
Darren Govoni, dgovoni@metadapt.com
(703)322-0384

Mike Alexander, mja@metadapt.com
(703)450-1005

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	29 oct 99	Final Technical 1 May 99— 29 Oct 99

4. TITLE AND SUBTITLE

Fractal Architectures & Complex Adaptive Systems

5. FUNDING NUMBERS

C) DAAH01-99-C-R118

6. AUTHOR(S)

Darren Govoni, dgovoni@metadapt.com, 703.322-0384
Michael Alexander, mja@metadapt.com, 703.450.1005

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Metadapt Design Systems, Inc.
11706 Tumbrel Court
Fairfax, VA 22030

8. PERFORMING ORGANIZATION REPORT NUMBER

SBIR-991-011A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

DARPA
3701 N. Fairfax Dr.
Arlington, VA 22203
Attn: Todd Carrico

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Findings produced in this final technical report may be reviewed by esteemed individuals from companies and institutions such as MITRE, DISA, BTG, Sun Microsystems (including Founder Bill Joy) and possibly others.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release.
Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

During this Phase I period we investigated the feasibility of software architectures that are inherently self-similar or *fractal* in nature. The reason for this is manyfold and include the current and future need to compose applications and systems on many scale factors using consistent metaphors. For this to occur, a uniform set of logical constructs and procedures that have a *quality of fractal* or invariance with regards to scale need be invented. To obtain such logical constructs, a clear distinction between the *logical* and *physical* aspects comprising a complex system necessarily should exist. This separation will promote greater fluidity between each aspect. Furthermore, the abstractions modeled in the software should be easily defined and visualized in a manner not tightly bound to physical constraints and requirements of a system. Furthermore, such logical entities should permit for the migration from sets of small to sets of large. This allows designers to operate in an additional dimension; that is, *designing in the large* and *designing in the small*. Although the concerns and implementations between levels can vary drastically, a consistent paradigm for understanding, creating, viewing and manipulating them would be a true leap forward in software and systems engineering.

14. SUBJECT TERMS

Fractal, architecture, complex adaptive systems, complexity, distributed systems, Dynamic systems, visualization, adaptable software, patterns,

15. NUMBER OF PAGES

71

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR

Introduction

This technical report represents the product of an SBIR (Small Business Innovative Research) Phase I investigative period lasting 6 months in duration. The original subject matter addressed by this effort is *Structures for Complex Adaptive Systems (Fractal Architectures)* and was part of the DARPA 99.1 solicitation offering (SB991-011). The goal of this initial research phase was to determine the feasibility of and approach to addressing the subject matter. We have formulated a somewhat broader *prime objective* to describe the exact nature and high-level objectives of this continuing research effort. As there are many interesting offshoots from the original subject matter, Metadapt is in the process of formulating a comprehensive vision through which separate, but related research efforts can be supported and will ultimately combine into a larger objective (code-named *The Gnomon Project*).

Research Summary

During the course of this Phase I effort, Metadapt has explored many areas relating to the prime objective. These areas include complexity, complex systems, adaptive systems, complex adaptive systems, natural systems, and fractal design among others. One of our objectives with this research is to accurately understand the landscape through which we intend to make important technological discoveries. The basis for these discoveries will come from many areas including computing, society, geometry and science. It is our hope that this continuing effort will shed light on many important findings.

Our immediate goal of understanding the application of fractal concepts to design explores the fundamentals of fractal geometry and the higher notion of general self-similarity. The purpose of which is to promote a better degree of human understanding and system growth potential while maintaining lower levels complexity.

The area of complex adaptive systems and its relation to computing systems is an important and growing field of interest in the community. Given current static and rigid software systems, it is an inevitable area of future exploration and will open the door to large evolvable computing systems. Coupled with this observation, our prime objective intends to control the complexities involved in building, understanding and adapting such systems by employing self-similar or fractal-like design constructs, procedures and behaviors.

From this initial feasibility study, we have drafted an initial approach to achieving the prime objective from a technical perspective (see *Technical Findings & Approach*). In addition to this, we foresee many key technologies and products emerging from this continuing research which we have also introduced (see *Software Tools*, *Commercial Concepts & Strategy*).

General Statements

With regards to our research obligations during this Phase I effort, there were no substantial difficulties or problem areas preventing us from arriving at this final technical report. All the major ideas and findings are included within this document (either explicitly or implicitly). With the arrival of Phase II funding, we are eager to continue with the developmental research and product commercialization.

How This Document is Organized

This final technical report is an evolution of the prior status report and therefore includes al pertinent content provided before including any alterations, modifications or deletions found to be necessary.

This report is organized into major, minor and sub-minor sections which provide and easy way to navigate the various concepts and sub-concepts discussed. The following list indicates the major sections contained herein.

- Introduction
- Research Summary
- General Statements
- How This Document is Organized

- About Metadapt
- Prime Objective
- Concept Findings
- Technical Findings & Approach
- Design Elements
- Software Tools
- Commercial Concepts & Strategy
- References

About Metadapt

Metadapt Design Systems Inc. is a newly formed corporation dedicated to inventing innovative technologies for complex adaptive computing systems and applications. Part of our mission is to uncover new paradigms for building complex systems and applications that are inherently adaptable, better controlled and understood. Through complexity visualization techniques, fractal architecture designs and a set of new power tools, Metadapt is setting its sights on the next century in computing.

Currently, Metadapt is funded via a small grant from DARPA (SBIR). Through increased awareness and market demand for new adaptive computing paradigms, Metadapt hopes to attract interested investors, scientists, customers, professionals and potential employees who are as excited about these areas as we are. Come visit us at www.metadapt.com and drop us an email (dgovoni@metadapt.com) or call us at (703)322-0384 or (703)450-1005.

Prime Objective

The *prime objective* of this research effort is to explore the potential for fractal-like designs and architectures that support large scale complex adaptive systems. Through its self-similar structure, such an architecture should allow for the dynamic composition of entities across scale. That is, the process of composing objects into patterns, patterns into architectures, architectures into systems and systems into larger systems should be as uniform as possible. Furthermore, the fractal qualities of an architecture should apply to other areas such as how applications and systems are composed, viewed, manipulated or otherwise understood.

The motivation for this type of endeavor is many fold. Within the military alone, the need for technology to meet dynamic and ever-changing demands is an ever-present and increasing force. Furthermore, systems are becoming more costly, larger and more complex as well. In consideration of these colliding forces, there must exist a way to allow these systems to meet the needs of a fluid and dynamic environment where time is of greatest importance and reducing the cost of maintaining and understanding such systems is essential as well. Therefore, these goals are recognized as part of our prime objective for this effort.

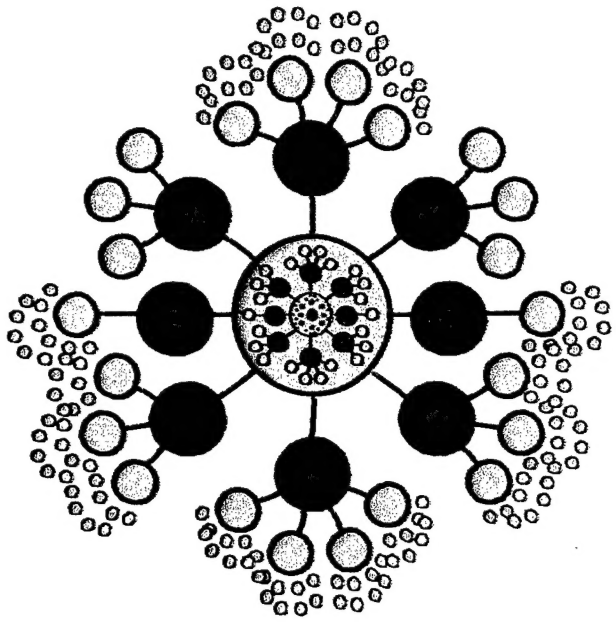
Another observed need is the ability to create and link systems together dynamically in a unified manner. Not only should we address the immediate concerns in the small such as assembling lightweight application components into hybrid tools on-the-fly, but also the need to create super-systems by composing existing systems in a manner that shields the higher-level structure from the structures within. Again, a common compositional theme should be prevalent at all levels whether creating a composite component out of smaller ones or a composite system out of smaller systems. Discovering, manipulating, modifying and adapting these structures should remain relatively constant across scale and complexity factors. This invariance to scale property is an important basis for a fractal architecture.

Components within this architecture should have the ability to identify their place and role within a greater system context through dynamic discovery and membership. That is, components should be able to come online anywhere on a network and dynamically become part of a larger functioning system; subscribing to and generating events and messages as well. Furthermore, such components should be able to relocate across a network without disrupting the running system, which may be using the component. Components will be able to dynamically discover and communicate with each other through a de-centralized, fault-tolerant registry service, which provides a solid basis for creating and deploying group enabled and workflow applications as well as complex dynamic systems. In addition to this, the architecture will allow secure access-controlled creation of event channels for objects to subscribe to. Such a publish/subscribe model is extremely powerful and necessary in large, dynamic and distributed environments [BattleField InfoSphere].

One of our conceptual innovations is negative-space, wherein deployed applications need not be infused with architectural framework code, but rather are deployed and tracked with negative space facets that provide this functionality transparently. Such technology will allow for the dynamic deployment and management of application services within the architecture while said applications remain free to implement varieties of existing technologies to accomplish their specific goals. This approach will permit the deployment of existing applications into the architecture with little or no modification.

A key theme of this effort focuses on *composability*. It is essential to understand the manner and method in which structure is created and ultimately composed. We wish to create a way to not only build applications with components on the fly, but also build, deploy and adapt distributed systems capable of evolving to great levels of complexity over time equally well. Regardless of the complexity factor, a unified approach to interacting with and creating objects, applications or systems will reduce the perceived complexity, time and resources needed to do so as compared to now. Our goal is to achieve this by providing the fractal organizational structures and negative space services within an architectural framework. Further into the future it may be seen that such

capabilities can be better provided in another manner such as a new programming language, environment or runtime containment service. Our shorter-term goals involve seeding this thought process and gaining valuable insight into the current and future feasibility of our objective. It is also worth mentioning that independent of the cacophony of technologies available to achieve some of our desired objectives, at the very least, the results of our research should provide a clean, implementation independent approach for what we believe will be a new paradigm for development.



FractalSphere
Drawing by Darren Gerard

Concept Findings

This section will introduce and discuss the major concepts brought together (thus far) by this Phase I research effort: *Fractal Architectures & Complex Adaptive Systems*. It is the intent of this portion of the report to provide a firm conceptual foundation necessary to navigate the technical accomplishments, design approach and future direction of this research. Furthermore, we have purposely designed our technical approach to be traceable from such high level concepts to their ultimate implementations. This offers a valuable degree of intentional forward separation between concept, design and implementation.

Please note that the research and findings on these issues are ongoing and certainly will continue to offer interesting and insightful discoveries related to the prime objective. As such, we hope to uncover much more than is contained in this summary document which represents only the initial study into this effort.

The following major concept categories (not limiting) are addressed in this section based on our progress into the feasibility of the prime objective.

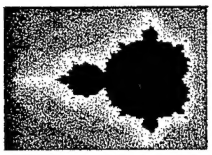
- Fractal Design
- Complex Systems
- Natural Systems
- Adaptive Systems
- Negative Space
- Fractal Worlds

We will not present the more academic side of every topic in great detail here, but rather will focus on the specifics that are most pertinent to our effort. Appropriate references will be supplied for further reading where applicable.

It is our belief that a major cross-section between these categories provides an area of important and vital discovery regarding software systems, complexity, design, and human understanding. It is our intent to illuminate as best we can the convergence between these major concepts into a core paradigm for creating highly scalable, complex adaptive systems. We will attempt to do this by discussing the concepts in some detail after which we will clarify our technical approach to creating a *lightweight framework* architecture that addresses these concepts. In addition, we will also provide details about available technologies and tools we will employ, design or otherwise develop to enable a solid and secure implementation of our designs as part of this and future projects.

Let us also express that the concerns and potential benefit of discoveries made during this Phase I are expected to be much larger in scope and include multiple ancillary technologies that contribute to the ultimate value and usefulness of the prime objective.

Fractal Design



Our ongoing research into *fractal design* will continue to examine the intrinsic properties of fractal-based systems in various disciplines such as geometry, nature biology, physics and others. It is the intent of this effort to rationalize the application of such properties to complex adaptive software systems such that the enumerated benefits to human perception and understanding brought about by such properties can be manifest in software systems. It is further the goal that the structural properties of systems across many disciplines can be realized and modeled to allow for equally complex, adaptive yet controllable software systems. From this, we hope to create a new manner in which systems (orders of magnitude more complex and productive than today's) can be readily designed, assembled and understood by humans. We expect to accomplish this by building mechanisms for constructing, visualizing and understanding architectures that draw from the design laws of the world around us as well as the way we perceive it.

It is essential before discussing the technical approach and design elements to clarify the basic principles pertaining to fractals and highlight the properties that we hope to extract and apply to designing systems.

Definition of Fractal

The term *fractal* was coined by Benoit Mandlebrot in the 1970's and is based on the notion of *fractional dimension* (or non-integer dimension). Although there are mathematically precise definitions of fractals, we will use one more suitable to our position: A fractal, in our words, is:

> **Definition 1:** *"a form or process that exhibits self-similarity with invariance to scale."*

What this essentially means is that the fractal object has many embedded levels of structure that resemble higher levels within itself. This is the most important fundamental principle underlying fractals and we'll talk more about it throughout this document.

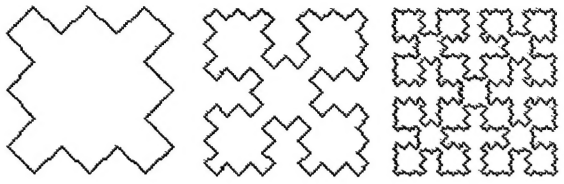
Quality of Fractal

Not only is a fractal a mathematical entity, but the term is useful to characterize something that bears the *quality of fractal*; that is, demonstrates an invariance of form or structure with regards to scale. And so, we can use the term to describe things like designs and architectures providing we have grounded the semantics of the definition which we will do in later sections.

Furthermore, let us state that it is inaccurate to label something as a 'fractal' when in fact it only bears fractal-like qualities. Mathematics is very strict about what a fractal object is and our intent is only to illuminate entities demonstrating qualities of fractals, but do not consider them to be fractals in the strict sense of the term.

Fractal Geometry

In fractal geometry, large geometric sets are produced by iterating through what is known as a *generating sequence* or *algorithm*. The process begins with an initial state and the generating algorithm is subsequently repeated *ad infinitum* to produce the geometric fractal set. These sets, often visually complex to the eye at large scales, bear an uncanny similarity in overall structure to their micro-parts or states produced by past iterations.



Sierpinski L-System surface filling fractal.

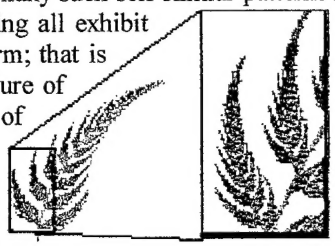
Iteration 1 **Iteration 2** **Iteration 3**

Fractals are seemingly complex in one sense yet quite basic and simple in another. Despite their apparent complexity, fractal sets can be succinctly represented by their generating formula and because of their inherent self-similarity, the *amount of* information needed to describe them is compressed.

Furthermore, from a human perspective, understanding the general organization and structure of a fractal set with inherent *self-similarity* should require far less processing than with non-fractal sets. Because fractal sets have a logarithmic period of similarity, the amount of information needed to describe the system will likely not vary as drastically from different magnitudes. It is indeed this self-similar structure that is most intriguing when studying fractals and applying their traits to our design efforts.

Self-similar Structure

The key characteristic of fractal systems is *self-similarity*, or invariance in form with respect to scaling [Schroeder 1991]. Indeed, we can see many such self-similar patterns in the world around us. Trees, mountains, clouds, lightening all exhibit similar structure and patterns throughout their entire form; that is to say at both the *macro* and *micro* levels. It is the nature of this scale-invariant structure that defines the essence of such things; to understand an entity's structure in this regard is to understand the entity. Indeed, nature employs this most incredible tool for generating and deploying intricately complex structures that appeal to our own innate sense of symmetry and beauty.



If we consider the application of self-similarity features to software architectures, we must find a medium through which self-similarity can be realized and subsequently become valuable to human perception and understanding. The domain of abstractions offered by programming languages is quite boundless, meaning simply that any number of discrete entities can be modeled (given properties and behavior). Therefore, it will be necessary to define discrete entities to use as fundamental constructs through which fractal organization and self-similarity can be realized. These objects will provide the necessary fractal composability forming the basis for constructing *fractaline* systems (i.e. systems bearing fractal qualities).

Much like iteration of fractal algorithms produces a recursively-defined structure, the byproduct of a software architecture based on inherently self-similar entities or objects should result in a system whose complexity (or description) can be decomposed recursively. The objective is a matter of organization and composability. The basic question being addressed is therefore:

> **Question 1:** *"How do we structure and organize objects such that the resulting organization is self-similar?"*

Ironically, the self-fulfilling answer to this question is embedded in the question itself, namely, "*...the resulting organization is self-similar...*" [We will revisit this later].

As is pointed out in the text Fractals and Chaos, "*self-similarity...is a property of the limit of the geometric construction process...*". However, this need not be a limitation in a highly abstract environment such as object-oriented software design. Self-similarity of these systems may also apply to the way or manner in which the system is used, extended, composed, or adapted in addition to being viewed and understood. This observation should permit composite objects and object clusters to be seen, manipulated and linked together in much the same way as individual objects with invariance to properties such as quantity, physical location, hardware configuration, operating system or other extrinsic properties [see section on "Physical/Logical Separation]. Specifically, not only do we see fractal composites the same way, we also manipulate them the same way. Indeed this is a

major characteristic of a fractal architecture. These observations lead us to the following corollary, which we believe has direct implications for the design of large-scale software systems:

> **Corollary 1:** *"If we want to build something invariant to scale, it should represent self-similar characteristics."*

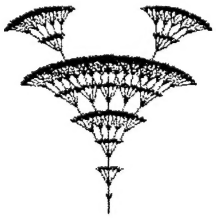
If this is true, then any attempt to create a fractal architecture paradigm should also include self-similar properties. However, before we can apply the notion of *invariance to scale* to general design, we must first understand what we mean when by the term *invariance* and the term *scale* in a context other than geometry.

Understanding Invariance

The notion of invariance we refer to in typical fractal geometry is most readily evident through visual interpretation. Specifically, the resulting geometry of a given fractal set exhibits *noticeable* structural repetition. The term noticeable is rather important here, as the representation of similarity can be either mathematically precise or asymptotic (where similarity is only approximated). Regardless of the underlying precision, such similarity is most evident through human perception.

Our goal is to explore the invariance of not just perceived design structures, but also other facets of software interaction such as use, behavior, properties and visualization. Let us also say that with regards to invariance of form, it may very well not be the case that design architectures are indeed constant across scale (or necessarily should be), for the macro and micro concerns of large complex systems will undoubtedly call for different design approaches. However, the quality of fractal is very much evident in that such stratified or embedded designs do exist across scale factors [see section on *"An Architecture of Patterns"*, *"Fractal Pattern Composition"*]. And so, in this sense, the invariance applies to the fundamental factor of composition which provides a uniform (invariant) way of composing patterns.

Scaling Characteristics



Another important trait of fractal sets is their ability to produce self-similar structures on an infinite continuum of iterations, or *scale*; that is, their *self-similarity is symmetrical across scale* [Cohen et. al. 1994]. Because of this, fractal sets and their subsequent geometry are in no way bound by the domain of number sets. This means that valid sets can be produced and iterated for an infinite number of values. In fractal geometry, increasing the number of iterations increases the computational complexity yet the visual complexity will not vary as significantly (depending partially on whether the underlying process is deterministic or random). After a certain number of iterations, a logarithmic period of repetitive structure will emerge (although not all of it can be seen). The *translation factor* of a given fractal set will represent the current computed range of iterations. This translation factor represents the scale of the computed set. Since the set of all possible values lies on an infinite continuum, only a portion can be processed and subsequently visualized and explored (also the spatial and dimension characteristics of your computer screen limit the amount of information that can be represented). Changing the translation factor corresponds to *zooming* in and out of a given fractal set (or, if you prefer, decreasing or increasing its scale respectively). Since fractal sets are self-similar, or invariant to scale, we witness the similarity of structure as we navigate in and out of the fractal set. This use of the term *scale* applies to visualizing an otherwise incomprehensible amount of numerical or spatial information. To indicate a specific scale is to govern the amount of information computed, viewed and interpreted.

Understanding Scale

Translating the meaning of scale into abstract design and programming language worlds is non-trivial. Terms like this are heavily overloaded across many disciplines and computer science is extremely laden with overloading of such terms. Therefore, it is difficult to discover much less decide on a precise meaning for just about any given term. Also, as technology changes, terms sometimes shift their meaning to compensate; however, we will recognize these situations when possible.

As was previously discussed, scale in fractal geometry is quite clear and precise. In order to satisfy our urge to define structures and call them fractal in nature, we must reconcile

the use of this term in that context. Some basic initial questions include the following:

Question 2: *"Can a design scale?"*
Question 3: *"How does a design scale?"*

It should seem appropriate that, since we are searching for ways to create designs with fractal qualities, that *the design* is the subject of our questioning.

In the natural sense of the term, most people will concede that large things have 'larger scale' than small things. That is, the scale property is somewhat independent of the entities involved and therefore relative to the perspective of such things. To look at microbes through a microscope involves things at a small scale, whereas to peer into distant galaxies involves large scaled objects. Of course, all of this is from a human-centric perspective. It is not difficult to follow this reasoning and apply some meta-property of size that correlates with scale and this may, in fact, be so. However, even size, must find an origin of some sort whether in numbers or through physical perception.

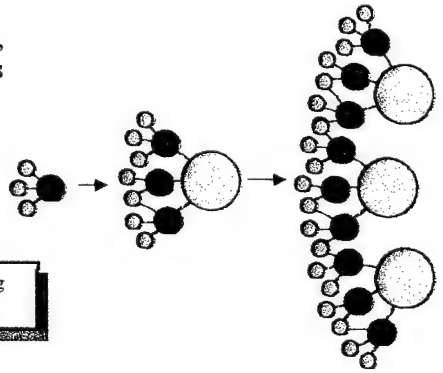
The difficulty is that something abstract like a design need not be rooted in mathematical precision and also need not contain a single concrete visual representation. In fact, designs in computer science tend to be of the highest abstraction and elude a consistent visualization. However long this has been the case, by their very nature, abstract designs can be described, represented and implemented and so have intrinsic properties that can certainly be represented in different ways including via visual metaphor. The trouble has been deciding on a single consistent way of looking at things, but we feel that in the future, visualizations can be created dynamically and tailored to properties of systems and designs that an analyst deems useful.

The use of scale in reference to design can certainly describe the number of entities present or embedded in a given design and therefore relate the notion of quantity, complexity or largeness with scale. Considering how the universe is built, there are very large scale things like galaxies and solar systems, that are themselves, composed of smaller scale systems and entities and so on, and so on. Using this simple translation, designs that embody other designs inherit a super-scale relationship to the embedded designs that is not explicit in either, but rather intrinsic to the very composition of the super-design.

To twist matters slightly, when we think about scale with regards to traditional software systems, we often refer to the complexity, size and resource demands required by the system. To say that a software system scales, is to say that it's basic design can accommodate an increase (or decrease) in resource demands and properly utilize available resources to meet those demands. In this sense, the term *scale* represents the size or complexity factor of such a system (including the amount of resources) and contributes to the systems overall performance.

In either case, the issue of scale deals with moving between the *large* and the *small*. Similar to how fractal structures can grow prolifically from small to vastly complex structures, so too should fractal based systems permit the fluid migration from small to large systems, utilizing the appropriate level of resources to meet demands along the way.

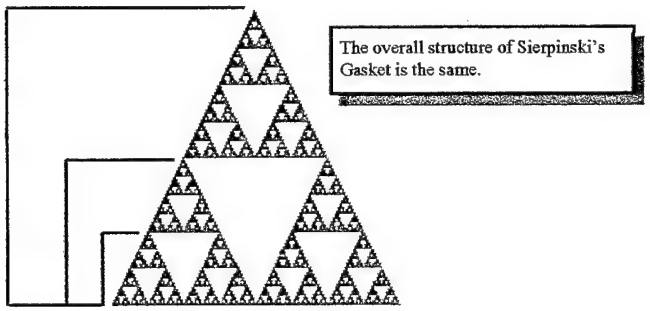
Finally, a fractal design is not a static design, but one whose scale-invariant property is manifest as the design grows over time.



A self-similar design exhibiting different iterative scale factors.

Perceived Complexity

An important issue when viewing or understanding fractal sets or designs involves *complexity* and *perceived complexity*. As we've pointed out, fractal structures maintain a similarity at many scale factors or levels within its geometric structure. It can be said from this, that the information necessary to summate the organization of the structure is minimized by its self-similar nature. In other words, its description at a *macro*-level would be useful (or reusable) also at a micro-level or vice-versa. This reduction in descriptive content parallels the overall perceived complexity of the structure. It is therefore a goal of our effort to allow for very large and complex self-similar structures to bear this quality as well.



If the goal is to allow for the dynamic creation of complex fractal-based systems, it must also be to manage that complexity in terms of how it is viewed, manipulated and ultimately understood. Complex systems are, by definition, complex and therefore beyond the ability of a single human to grasp in their entirety. Let us then consider this observation:

Observation 1: *"The benefit of applying fractal design methodologies to complex software architectures is to gain the benefit of complexity management and understanding through self-similarity and scaling. "*

If such a system beyond comprehension is to be wholly or partially viewed, manipulated or understood, it must provide the appropriate amount of information for the given scale factor. The basic idea is that such a system would be best organized *fractally* (exhibiting self-referential similarity). This organization permits cross-sections of specific levels to be isolated. When viewing entities within the system, we are not forced to visually process all embedded structures and elements within those entities. By maintaining a *constant amount* of information required to describe a particular location, scale factor or level within a system, humans can better digest the complexities of the overall system.

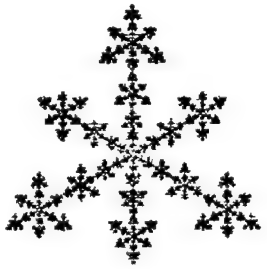
Zooming

The invariance to scale of fractal sets can be valuable when applied to the visualization and construction of massively complex and distributed systems. Since enormous amounts of information surpass human capability, the necessity of a "zooming" mechanism becomes apparent. Similar to how one explores a fractal set and understands it's various undulations and similarities, we believe the structure of complex object systems including (but not limited to) it's recursive organization, purpose or topology can be viewed, manipulated and better understood. Providing the ability to prune unnecessary details in the visualization of a system as well as provide the ability to view different cross-sections or logical mappings of that system will provide an unparalleled ability to create, manage and understand vast complex adaptive systems [see section on "*Complexity Visualization*"].

Generating Systems & IFS's

Geometric fractals are produced via mathematical processes (e.g. functions or random numbers) and iteration. IFS's, or Iterated Function Systems, provide a method for constructing complex fractal sets based on a simple repeated form, or fractal. Through affine transformation and frame placement, deterministic fractal sets can be systematically constructed. The process that constructs very large fractal sets is often quite simple and elegant and much more so than the beautifully complex fractal it produces. The seemingly paradoxical notion of something elegant and simple generating something intricate and exceedingly complex is an interesting focus of our research.

When we consider the use of fractal properties and generating functions in an object-based distributed system, we must provide, again, an appropriate mapping between the steps and terminology between the worlds of math and computer science. We suggested earlier that designs and micro-architectures have an implicit form based on their composition and thus can be represented isomorphically in a geometric space. Connecting *object* groups and patterns together to create larger aggregate architectures is one concept in fractal architecture construction we will visit later [see section on "***An Architecture of Patterns***"]. For now, it is sufficient to suggest that the architecture of a large object system can be defined, captured or otherwise described and reproduced using something similar to an IFS in the math world. In the case of objects and patterns, they can be deployed remotely and connected into an existing collective thereby increasing the complexity and structure of the overall system in a stepwise, adaptive and iterative manner. Because of the iteration and repetition of micro-patterns and object groups, the larger context would exhibit self-similarity in much the same way as IFS fractals would. [see section on "*Mapping boundaries & form*"]



An IFS generated fractal snowflake
Generated with Fractint for Windows

Architectural Synopsis

The introduction of the major characteristics of fractals and fractal geometry lead us to the following conclusions about their applicability to design.

1. **Self-Similarity is key to a fractal based system.** Since the notion of self-similarity limits the amount of complexity or variance of structure with regards to scale, constructs designed on this principle should exhibit similar traits. The goal then, is to provide a meaningful mapping between the terminology of scale and invariance from its geometric origins into other domains such as software design and systems architectures.
2. **Fractal systems cannot be seen all at once.** Because of the infinite continuum of values produced by fractal sets, they are too large to be seen or comprehended at once. One can also claim that a sufficiently large complex system (like, say, the Internet) is equally unviewable in its entirety. When a given system of components (such as objects) reaches incomprehensible proportions, it is necessary to provide a method of scaling through the system such that the structure of it can be decomposed, understood and ultimately manipulated.
3. **Fractal systems should grow gracefully.** Given a simple system, it should be permitted to rise in complexity in a (relatively) boundless manner without compromising its basic structural design or architecture.
4. **With fractal systems, there is a gap between actual and perceived complexity.** That is to say that the amount of information needed to describe a given fractal system may be reduced through repetition of structure. This property is most evident through visualization. Using visualization and interaction techniques we want to prune insignificant or repetitive details of a given system and allow its inherent similarity to speed human understanding.
5. **Fractal systems can be iteratively designed and generated.** As we see how fractal sets are produced through iteration of a generating process, we hope to provide a manner in which fractal software systems can likewise be generated. The manner in which such systems are to be generated might involve taking a *descriptive string* [Bar-Yam 1997] representing the property state of the system or a portion thereof and subsequently creating, instantiating and deploying it. Likewise, the description of a given system instantiation could be acquired by traversing its components and decomposing their structure and properties and storing it in such a descriptive string [see section on "*Describing Complexity*"].

Complex Systems

A complex system is one that involves multiple interacting elements. Because of the number of permutations of state of a given system changing over time, complex systems are inherently difficult to understand, describe and predict. For example, the weather is classical example of a complex system (yet still deterministic from the reductionist perspective). Such systems are extremely non-linear and therefore subject to chaotic behavior. Fortunately, discrete software systems are designed not to exhibit the levels of unpredictable chaos found in nature, but they are definitely getting more complex and we must find ways to manage and understand that complexity nonetheless. Attempts to harness that complexity via a fractal architecture warrants a brief discussion on complex systems and complexity.

Complexity, Systems & Events

Complexity arises when the number of interacting elements is large. Complexity is difficult for humans to understand, as the permutations of cause-effect relationships soon become overwhelming to compute or predict due to their non-linearity.

Complexity is not so much a physical property of a system, but rather a manifest property present as we humans find it nearly impossible to grasp or calculate the sum of interactions present in such a system. Thus, it is from the perception of humans that we baseline our definition of complexity. So when we say a system is complex, it is posed in terms of how much information the human mind must process consciously in order to understand (or model) it. To this effect, all systems of interest to humans, including software systems can be rated in terms of their apparent complexity to humans.

Given this perspective on complexity, let us explore the idea of a complex system. A system involves the participation of a variety of entities (sometimes referred to as elements or particles). The complex systems we are interested in typically contain a large number of elements, many (if not all) of which can interact with one another via discrete events.

In physics, an event occurs when energy is transferred between two particles of matter. This transferal can take many forms such as radiation (in the form of light), kinetic energy, electromagnetic etc. When such a transferal occurs, we would model it as a discrete event in time. The presence of an event causes a state change (in the universe); and, as events in the universe are in constant motion (as indicated by the second law of thermodynamics), the universe is continually changing state. Most complex systems, by their very definition, are also changing state. The rate of change over time is governed by the propagation of events through the system. Elements within the system can generate, absorb or reflect events. Therefore, interaction between elements occurs through the exchange of events; elements may change their individual state upon receiving an event or create a new event. Therefore, elements may generate and/or absorb events [we will spend more time discussing components and events in the Technical Approach sections on "*Distributed Component Architecture*" and "*Event Communication*"]. It is indeed, through such generation and exchange of events that elements or components in a system have an opportunity to *react*. Only through such reaction over time will a system exhibit the behavior it was designed for.

Describing Complexity

As is pointed out in the text "Dynamics of Complex Systems" [Yar-Bam 1996], complexity is defined as "*..the amount of information needed to describe something.*" From this, one can conclude, quite simply, that simple systems will require much less descriptiveness to represent their running state than more complex systems.

Independent of the type of system being described , the description must be isomorphic to the system it represents. That is, there exists no degree of separation between the state of a given system and it's descriptive representation at any given point in time. In order for this to be true, an appropriate mapping between such a description (referred to as the *descriptive string* [Bar-Yam 1997]) and the system whose state it captures, must exist (as well as a way of moving between the two).

The question then arises:

Question 4: *"How do we capture the state of a complex system?"*

The answer to this lies in the ability to traverse and decompose elements in a system. Inspecting each and every element and acquiring its state (called *state decomposition*) provides the sum of all states of elements in the system and hence the system itself. Therefore, to speak of a software system or architecture based on connected discrete objects permits us to capture its state by traversing its *attributed graph grammar* [see section on *Attributed Graph Grammars*] (regardless of where elements of the system might be located over a network) [see section on "*Logical/Physical Separation*"].

Regarding our objective of creating a lightweight fractal architecture, it seems to follow that if elements within the architecture have self-similar attributes, that a proper recursively defined algorithm for traversing the structure of a fractal system can be hence produced and the state of such a system captured. It is this state that can be used (among other things) to represent the complexity of a given system. The format of this state is not necessarily vital, but may take many implementations (e.g. XML); however, the resulting grammar should be self-referential.

Complexity Visualization

Visualization has enormous benefits to human understanding, and the advances made in information visualization are a clear indication of this. Currently, in software systems and architectures we design and blueprint them initially and, like conventional architecture, these massive structures tend to remain as they were built (or change very slowly). However, if we are building inherently adaptable systems, they will tend to deviate rapidly from their initial design state, and perhaps design documents as well. It should be possible, then, to understand the evolved topology, organization, or architecture of such a system dynamically as well. Not being able to do so would likely cause confusion about its current state versus its initial design.
From this realization, we will find the need for a new breed of visualization tools that are able to peer at a live complex software system and display it in a variety of ways and dimensions. Some of these might be the way such a system is constructed with its object components, connections, fractal geometry and structure on many levels.

Such a tool would operate on a system while it is running, to give a dynamic and accurate view of how it is composed and operating. Furthermore, this new breed of tool would permit for the interaction of component objects as well, which provides a uniform and dynamic way to understand complexity as well as to advance it via visual feedback.

The need for good software visualization tools has long been recognized, but so has the difficulty in constructing them. Frederick P. Brooks (Professor and founder of CS Dept. at UNC Chapel Hill) has this to say on the subject of visualization and software systems [Brooks 86]:

> *"As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs superimposed one upon another."*

He goes on to add:

> *"In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, and thus do not permit the mind to use some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds."*

Although Brooks felt that such efforts (at the time of his writing) were unfruitful, we believe that new discoveries and techniques we hope to pioneer in fractal architecture will

pave the way for breakthroughs in this exciting new frontier. Specifically, we feel the following corollary to be true:

Corollary 2: *"Any system with an underlying representation can be modeled and visualized."*

The term *representation* is key here. It is a fact that any system, by definition, has some form of underlying representation. For example, a pool of water contains countless water molecules. Nature already gives us an easy way to visualize such a system (namely radiation and the laws of physics). Modeling such a large system is not impossible, but requires enormous amounts of computing power as well as the ability to determine the state of elements in the system. In doing this, any type of visualization of said system can be realized. Software systems are no different. They contain a finite amount of information arranged in various ways. Because of this, there may be many types of views into that system that are useful. Determining what to extract and visualize is a first critical step. Our framework will provided the basic structural and architectural form from which useful structural visualization can occur.

Architectural Synopsis

The following list highlights some extractions from our ongoing research into complexity.

1. **Elements or components in a system exchange information through event interaction**. That is to say that participants in a system can either *generate*, *reflect* or *absorb* events and therefore originate or propagate change consistent with the laws of thermodynamics in the natural world. For computer software types of systems, change is a simple byproduct of its operation and therefore without change, the system may be in a wait state or otherwise inoperable.
2. **To control the descriptive content is to manage complexity.** Although complex things are inherently so, we can make them easier to understand by controlling the amount of information or descriptive content we must digest at any given moment. If such content cannot be viewed all at once (due to its size), it must be broken down. A complex system that has embedded levels of concern or structure can better map it's descriptive content in this manner and hence control the amount of which need be displayed and subsequently processed at any given time.
3. **All systems have some form of intrinsic representation and upon extraction such representation can be visually modeled**. The key to future visualizations of complex systems lies in the ability to extract meaningful properties from a system dynamically. With older generation computer languages, properties within a program are more or less lost after compilation. With the presence of advanced OO languages like Java that host a runtime environment, inherent and meta properties about objects can not only be extracted or inspected dynamically, but can also be assigned or changed dynamically as well. Once this capability exists, the path to useful visualizations lay ahead.
4. **Visualization is key to understanding complexity**. Whether viewing megabytes of numerical data as color-coded three-dimensional terrain or thousands of interconnected and interacting objects and components in a running system, most of what we'll understand will enter through our eyes. Getting the most out of our visual bandwidth will involve taking advantage of many dimensions of data that can be easily fit onto a small two dimensional viewing surface (e.g. color, size, proximity, temporal as well as spatial aspects and others). Furthermore, from the realm of fractal geometry, the most enjoyable as well as informative periods are during the exploration and viewing of fractal sets rather than the numerical analysis (or review of design documentation or source code during systems development).

Natural Systems

In order to better understand the application of fractal architectures to complex adaptive systems, understanding the structure, behavior and complexity of such systems is important. If one wanted to learn about complex adaptive systems, there is no better model than the world around us. The natural systems we are a part of and that surround us epitomize the concept of complexity and adaptation. Therefore, we will focus a small part of our research on finding the valuable qualities of natural systems and how they may be represented or supported through a fractal architecture paradigm.

Notable Characteristics

Nature is known for its ability to survive through adaptation. Darwin's fundamental notion of "*survival of the fittest*" is based on a few basic observations about living things and their relation to the environment. Specifically, the environment changes and through such change comes forces that reward or select genetic traits for survival. Living creatures must keep up with changes in the environment through successive adaptation across generations and hence large populations exist to ensure the survival of the species as a whole.

Although this is a well researched subject, it is interesting to identify ways for object-based software systems to undergo successive alteration in a controlled and deterministic manner. Since nature has ensured that living things are equipped with the ability to acquire and pass on successful gene sequences, there does not need to be an outside controlling force or entity to administer such changes. In something like a software system, the components are very discrete and not subject to randomness or mutation in the same manner as living systems. This is not to say that such a discrete system cannot be imbibed with the ability to govern its own adaptation and mutation, it's just that we believe, initially, that human-centric methods will provide the most fruitful and interesting results. However, we do believe that *self-governing systems* will be a useful future research effort and an important technology of the future [see section on "*Self-governing systems*"].

Populations & Predictability

With complex adaptive systems, there exists a population of entities. Whether water molecules flowing down the path of least resistance or domain objects in a large distributed system, the complexity is governed by the population. In natural systems, there is typically free-form interaction among entities which are all "glued" together by the laws of nature and physics. However, in a compiled software system, things are built a certain way and continue to operate the way they were designed (whether successful or not). In other words, there is a high degree of predictability in software systems inasmuch as it is fully understood and its environment remains relatively constant. This is not so in natural systems and we do not necessarily want to find ways to model the inherent chaos in natural systems. Rather, we would wish to understand the dynamics of populations and their collective behavior and apply useful heuristics, metaphors, or designs to software systems composed of multiple discrete, autonomous or otherwise connected and interacting objects/components as well to gain the quality of fluid adaptability in a changing environment.

Ant Colonies

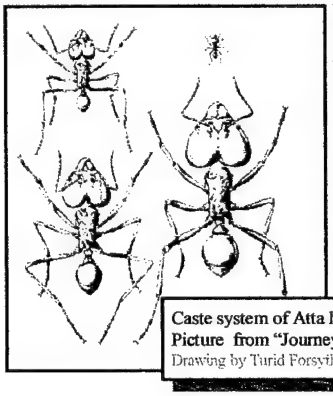
One very interesting approach in our study into complex adaptive systems focuses on the structure and behaviors of ant colonies. Ants are one of the most successful species on this planet and their lengthy history is living proof. Many terms arising in ant research sound strikingly familiar to current discussions in distributed systems. Some of these phrases include "*no single point of failure*", "*redundant or fault-tolerant facilities*", "*dynamic*", and "*reactive*" to name a few. Therefore, as part of our ongoing understanding of complex systems, we will continue to pursue a greater understanding of the success of ants.

One of the defining characteristics of ant colonies is their ability to function as a collective. In order for the colony to be effective, it must be able to respond and adapt to

forces driving it out of equilibrium. The sheer size of ant colony populations allows it to accommodate many simultaneous and disparate forces it may encounter. For example, at any one time, situations such as the following may occur: an intruder has breached the colony or the colony is under attack; the pavilion where that ants live has become crowded and needs modification and expansion; or food is in short supply and more effort must be put towards replenishment. Because of the constant needs of the colony, ants are always busy and attending to matters. The seemingly chaotic turmoil we see when watching ants scurry about is actually part of why they are so successful at reacting to these forces. One of the notable qualities of ant colonies is how different types of ants play different roles within the community.

Caste Distribution

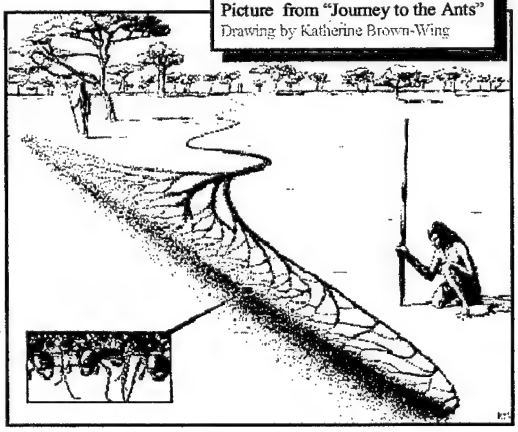
The *caste distribution* of an ant colony refers to the types (or *classes* if you will) of ants and their duties with respect to the entire colony. For example, there are ants that perform certain specialized duties such as forage for food, build structures, feed larva and defend the colony. Many ants perform more than one kind of duty on demand. This distribution of role is vital to the functions of the colony as a whole as it permits the specialization of form and function for ants within the caste. For example, worker ants have larger heads and mandibles for carrying food, tearing leaves and building structures. The queen on the other hand has a form useful in bearing offspring — many thousands of offspring.



Caste system of Atta leafcutters.
Picture from "Journey to the Ants"
Drawing by Turid Forsyth

The Superorganism

The popular adage "*The whole is greater than the sum of the parts*" applies quite fittingly to ant colonies as well. The higher-order behaviors of the colony are driven by the combination of *internal* and *external* forces. No single ant contains an exact blueprint for behavior exhibited by the colony proper. Collectively the ant colony shows behavior, structure and form unique only to itself. So much so, that some entomologists refer to it as a *superorganism* [Hölldobler et. al. 1994].



Driver Ant superorganism.
Picture from "Journey to the Ants"
Drawing by Katherine Brown-Wing

Emergent Behaviors

The behaviors of the ant superorganism can best be characterized as *emergent behavior* as it arises not from the direction or function of a single ant, but from the combination of micro-behaviors exhibited by the ant collective at a macro level. Nature has so optimized the reactiveness of ant colonies that their ability to respond to immediate concerns is quite astonishing. If an intruder such as a wasp invades a colony, the first available ant will attempt to eradicate it. Nearby ants witnessing this will respond until the enemy is defeated. Interestingly, however, the colony only responds with exactly as much internal force as is needed to perform the function. In other words, ants will get involved *only if needed.* This holds true for food gathering, building and other duties. This optimization permits the colony to be functioning at its highest capacity in order to address numerous concerns that need constant attention. If more ants responded to an event than were needed, the efficiency of the colony would suffer and its ability to handle simultaneous concerns would be diminished

This capability is quite similar to the notion of *quality of service* in networking and computing. An important aspect of Quality of Service (QoS) is the ability to respond to dynamic needs of a given system according to availability and demand. In networking, QoS provides bandwidth availability for those services that require it when they need it. Otherwise, extra bandwidth is consumed and distributed evenly among equally competing services. Therefore, a QoS system can react and adjust to external demands as they arise. This is an important concept in our research efforts.

Embedded Structure & Purpose

Ant colonies are not simply an enormous number of ants running about in a uselessly chaotic manner. Groups of ants arise to accommodate conditions within the colony as we noted. These so called *teams* [see *The Ant Fugue*: Godel, Escher, Bach; Hofstadter 1979] will assemble spontaneously and decompose when the objective is completed. Resources will rise and fall based on need. Given this, the organization of an ant colony at any one time may be very complex with multiple levels of teams responding, reacting and collaborating with each other. Individual ants may unknowingly pass from one team to another and teams may pass through each other like ghostly galaxies.

This embedded structure, where teams assemble into larger teams, is a form of self-similarity not so much in the sense of geometry, but in the sense of function and purpose. We can therefore look at a colony and understand it's purposeful behavior and organization as a recursive structure with the colony *superorganism* at the top and an individual ant at the bottom. This is a fractal quality and although such a visualization may surely never exist for ant colonies, we hope that software systems can be built, adapted and understood with the advantage of visualization, the necessity of which should be clear.

The Architecture Colony

Let's consider for a moment an enormous software system composed of thousands or even millions of objects scattered across vast distances over networks. Such a system would surely not be born overnight and would need to undergo periodic growth and evolution along the way. For such a system to be rigid, static and bound to its environment would result in long periods of down time and ultimate extinction as technology raced ahead. Therefore, such a system should be flexible and adaptable (allowing for dynamic and fluid enhancements over time). System requirements, resources and external forces will be forcing such a system out of equilibrium, and favor. When this occurs, users will notice degradation in performance and value. Furthermore, someone will need to address the problems arising in the system and since there are millions of objects geographically disperse, it is a logistic improbability (using current industry technology) that any such problems would be fixed soon. Therefore, such a system should allow for viewing and understanding of its logical representation from any and many vantage points (including geographic location). In addition, the fractal structure should be easily traversable by humans and automated processes alike. If this is so, then interacting with such an enormous system can be accomplished easily through its visualization. If one can reduce the complexity of such a system to an iterative process of scaling, then it can be understood and more easily fixed, maintained, and adapted. If this is true, then such massively large adaptive systems will be economical and beneficial to humans and hence their existence will be justified. The following corollaries seem to fall out of this:

Corollary 3: *"In order for large complex systems to arise, they must be economically justified."*
Corollary 4: *"Before economical justification occurs, technology advancements must be made."*

Self-governing Systems

We talked a bit about natural systems such as ant colonies and previously explored a scenario involving a massive distributed system. The self-governing capabilities of ant colonies is intriguing and has influenced the way in which we think about systems design during our research. The notion that a complex adaptive system can govern its own (human specified) equilibrium levels and make rational adjustments based on threshold values is an interesting area that we wish to explore in future research efforts. Essentially, we see that in the future, systems of enormous complexity will need to

govern their structure and performance thresholds in order to maintain optimum performance and availability, and that this attending-to must be performed in a distributed manner rather than in a centralized one. By their vary nature, these complex beasts will be difficult for humans to constantly attend to. A heuristic-based scheme for controlling change and evolution will allow such a system to seek its ideal range as it continuously grows, scales and evolves. Some possible actions might include coalescing distributed objects onto newly available resources, watching message traffic and optimizing object proximity, creating pools and clusters of objects on demand, managing quality of service features, etc. We hope that future SBIR R&D efforts will permit such research to augment our current exploration into complex adaptive systems and fractal architectures.

Adaptive Systems

Complex systems in nature are fluid, graceful entities that migrate over the landscape, propagating and evolving. Key to such systems is their ability to adapt and so to study complex adaptive systems, is to also study *adaptation.*

It should be clear that large complex software systems that are rigid become obsolete fast. Marshal McLuhan phrased it best when he said "*If it works, it's obsolete.*" Software systems of the future will be vastly more complex than today's and will be so because they are adapted and evolved incrementally over time. Nothing large and complex is born overnight, much less understood in the same time. Current systems take so long to develop that indeed their usefulness is often surpassed during that period, and are constantly trying to catch up.

Overcoming Obsolescence

One of the primary motivations for software adaptation is to keep pace with changing technology. The motivations for software systems today are based more on human needs and human processes than on the basic capabilities and limitations of the hardware. These needs are constantly changing and advancing, and software systems must be able to adapt to this change. The demand for systems that can be adapted quickly and easily without re-implementing will only increase for these reasons.

Manipulating Different Levels

An important characteristic of adaptive systems, we believe, involves the inherent quality of levels of representation in the system's general structure. That is, the complexity of a given system can be shielded through various embedded levels of descriptiveness that reflects the underlying architecture of the system (i.e. it's form). The designer of such systems can isolate levels of concern and impose and manipulate structure on those levels. Through visualization of such levels, a complex system can be digested or constructed in discrete portions. Each portion within a given level can have properties and interactions and thus can be manipulated or adapted as well. Being able to define, view and manipulate levels of containment as well as the entities within them is an important quality for a complex adaptive system, if such a system is to be properly evolved and understood by human designers. The embedding of levels and structure is a form of self-similarity.

Software Should be Soft

In order to provide needed functionality, system architects use a combination of hardware and software to build information systems. By design, these systems are typically layered, with the hardware filling the lowest level, followed by various layers of system software, and finally application software, like so many layers of sediment. Clearly, one of the reasons for stratifying information system architectures, and for using well-defined interfaces to define the boundaries between layers (e.g., application programming interfaces, APIs) is so that the implementation of individual layers can be changed with no (or hopefully, minimal) impact on the adjoining layers. For example, in the 1980's, many developers discovered that by coding their applications in the C programming language and sticking to a core set of GUI and OS API's (e.g., X-Windows and Unix System V), it was relatively easy -- although not trivial -- to move code from one Unix environment to another simply be recompiling it. Similarly today, application developers typically code database applications to an ODBC or JDBC API, which allows the implementation of the underlying database to be changed with relative ease (e.g., upgrading an ODBC application to an Oracle database).

This application development paradigm -- coding to well-defined open APIs --is well-known and has become, in effect, the staple approach to modern large-scale software development. This approach lets developers "upgrade" information systems by swapping in (generally during the development cycle, but also possibly at runtime) new implementations for layers.

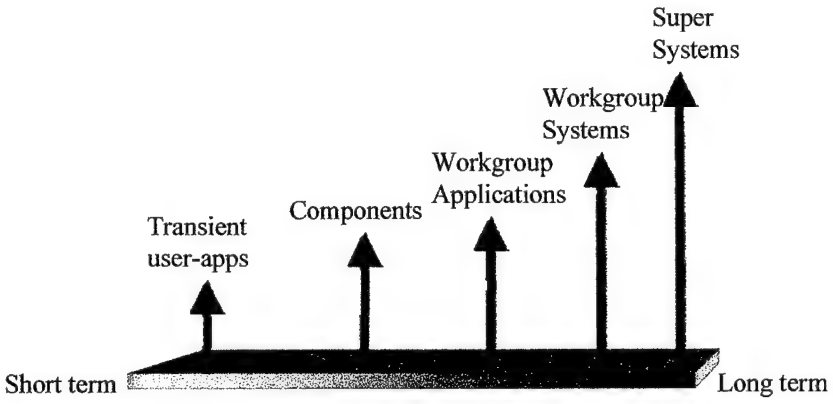
Although this approach is powerful (the alternative of not coding to well-defined interfaces harks back to pre-structured-programming days), it still contains a number of drawbacks. The major drawback of this approach is that it is fundamentally a mechanism for upgrading the implementation of components/layers of an architecture, rather than upgrading the architecture itself. That is, the use of interfaces/API's localizes the effects

of changes in the underlying implementation, so that, for example, a new or modified implementation of a component/layer can be substituted for the old one, as long as it too conforms to the original interface/API. However, in an ideal world we would like to be able to change not only the implementation of an interface/API, but also be able to change that interface/API itself.

Long-term versus short-term Adaptability

Just because a system or application has adaptive qualities that allow it to continually change does not mean it will need to do so. It should be emphasized that although change is constant and inevitable, the *ease of change* is more important than simply the *ability to change.*

Systems may in fact undergo infrequent adaptation from a relative view. However, when systems need to grow and evolve, they must be able to do so as painlessly as possible. Applications on the other hand might undergo frequent adaptation as new services, components and capabilities are incrementally available. Furthermore, users may use a variety of components to assemble short-term or transient types of tools and applications. Some of these components may very well be pulled from existing applications to create scaled-down or specialized versions. Therefore we see that on a continuum lie short-term or immediate concerns at one end and longer-termed ones at the other. Likewise, we might find that smaller tools and applications gravitate towards shorter-term needs while large complex systems gravitate towards long-term needs.



Negative Space

Negative space is a new concept that is useful for understanding and constructing next-generation distributed-object systems. Negative space is a new way of thinking about -- and building -- scalable distributed-object systems. Like other paradigm shifts, such as that from unstructured to structured programming and the later shift from process-oriented programming to object-oriented programming, the concept of negative space may seem at first rather elusive. It is a new way of looking at distributed systems, and we believe that once the concept is grasped, its value becomes apparent and that it becomes difficult to think of distributed systems in any other way.

Let's look at this from a somewhat historical perspective. Object oriented (OO) programming has a long history, one which pre-dates the development of distributed systems. The focus of OO programming is "objects"; objects send messages to one another, and these messages can be thought of as events of which objects can request to be notified. In a non-distributed system, the delivery mechanism for events is relatively straight-forward, at least conceptually. An object that wishes to be notified of the occurrence of an event has one of its methods, a listener method, invoked when the event occurs. In essence, an event notification becomes, using process-oriented programming terms, a function-call invocation. Although we "know" that this translation (form event-notification to function-call invocation) is occurring, a significant part of the OO paradigm shift comes from learning to think of this in terms of event-notification rather than function-call invocation.

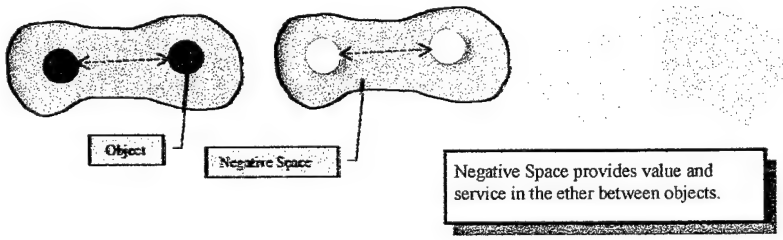
Things get a little more complicated for distributed systems; in today's systems, the programmer must be aware of where objects are located, either locally or remotely. This is apparent in today's client-server paradigm; objects are created to operate locally or remotely; e.g. the server "knows" that the client is located remotely, and similarly the client "knows" that the server is located remotely. For example, middle-tier objects typically talk to one another differently than do clients that talk to middle-tier objects. Unlike in our non-distributed OO systems, where the message delivery system was uniform (because all objects were resident in the same address space), in a distributed object system the delivery mechanism is non-uniform in that clients must be aware of the remote nature of the object during initial acquisition. Furthermore, from a design perspective, it typically must be decided which objects are to be remote enabled and which are not. Specifically, the role of objects is determined at compile time; certain objects are typically made remote, others local. If at a later date, an object that is remote needs to be made local (or vice versa), the code must be changed in order to accommodate the changed event-delivery mechanism that will be used. From a pure application-logic perspective this is unfortunate; the business logic has not changed, only the location of that logic has changed (e.g., it moved from client to server, or vice versa). From the application-logic standpoint no business logic has changed, yet the "code" must be changed. Ideally, we would like to make these objects --and their implementation -- location independent. If that object needs to be deployed locally, it should be able to be deployed locally; if later it needs to be deployed remotely, that *same object* should be able to be deployed remotely, without requiring any code changes. That is, the event delivery mechanism should be completely separated from the objects themselves. This is a central notion behind negative space.

Where is the negative space?

The internet has brought a new level of interconnectedness to the world of programming and applications. It is the interconnectedness from which value is derived in the network economy-- this is Metcalfe's Law. One approach is to view connections as objects also; this follows almost immediately from the OO perspective, where everything must be an object. We ask if this approach is optimal, or is something else missing?

Our approach looks at this landscape of components differently; whereas the object community sees things largely in terms of objects, we see the world as being composed of objects and a *medium* in which they are embedded. In architecture and design, these two classes of "spaces" are called the foreground and the background; or the positive space and the negative space. It is our contention that the object community has thus far focused

only on the positive-space world of objects, and has not recognized the power of treating interconnections as occupying the negative space.



Where does negative space come from?

Rather than focusing on the underlying hardware and system software, in an interconnected world it makes more sense to focus on the business logic you can provide, letting the system software and interconnectedness be "out there" from the business-object's perspective, rather than "in here".

Our business objects tap into the network through a TCP/IP stack; the business objects can be viewed as floating in the negative space. The network is the negative space.

Why is negative space important?

Negative space permits the separation of application-level functionality from system-level functionality. Although we all try to do this -- the OS and/or JVM are separate from our business logic -- nevertheless, traces of system-level functionality are all-too apparent in applications these days. For example, when we code remote objects in Java, we must specify that they descend from Remote, UnicastRemoteObject, or other base classes or interfaces which explicitly characterize the remote property of an object. (For example, the term "Remote" is a computer-science term, not a banking term.)

For years, the computer science crowd has drawn the Internet as a cloud; packets go into the network/cloud and packets (miraculously) arrive at their destination. As users of TCP/IP, for example, we have no idea how our original message, which was broken down (and subsequently reassembled) into datagrams and frames during delivery, actually arrived at its destination. The message may have traveled through fiber-optic cables, perhaps even bounced off a satellite, before arriving at its destination.

This is one of the beauties of the Internet and in particular its underlying transport mechanism, TCP/IP: application software is shielded from low-level protocol specifics and underlying network technologies by TCP/IP's layered model. By this measure, to an application program the transport mechanism for TCP/IP can be thought of as "negative space".

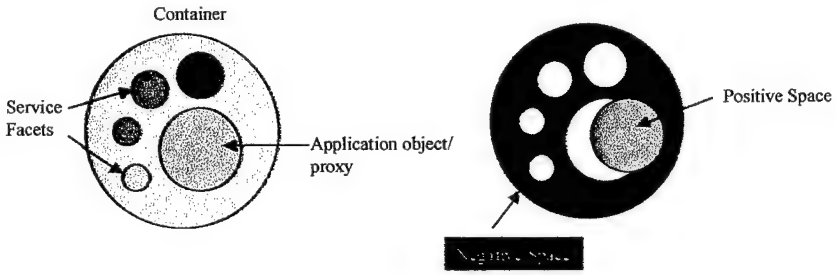
This approach makes a lot of sense for programs that communicate with one another via sockets; we ask whether an analogous concept of negative space ought to be applied to the delivery of object-to-object messages. That is, objects should operate at the level of messages and events; how these messages and events are delivered (i.e., the "protocol") should be shielded from the objects. Much like when we pick up the phone to relay a message to one of our colleagues, we are generally not concerned whether the message will be carried by AT&T, Sprint, MCI, etc. (In fact, these companies lease bandwidth to one another, so it might not be so clear who exactly is actually relaying your message.)

Technology changes quickly, and we do not want our systems to become obsolete at the rate at which technology advances. Unfortunately, using today's software development methodologies (e.g., business objects are coded-to-CORBA, coded-to-RMI, etc) this is generally the case. Ideally, we would like to be able to have objects communicate using whatever inter-object communication mechanism is currently best, be that CORBA, RMI, or something else. Only when this communication mechanism is part of the negative space will this be possible, where business objects will use the negative space as a service that simply delivers the specified functionality (much like TCP/IP simply delivers packets).

The negative space deals with the *stuff* between objects meaning that the objects themselves can be largely unaware of how the negative space is implemented and will not notice even that it exists. From the standpoint of objects and inter-object communication, the world appears to be all objects, which is what OOP offers (and may also be a drawback). Special negative space objects called containers and facets will provide interesting and useful implementations of key services needed in a complex adaptive system. However, the presence of such services will not appear foreign or otherwise unique from the standpoint of application objects. In some cases, application objects will not be aware that they are, in fact, communicating indirectly to their targets through negative space facets that perform critical services such as *store-and-forward messaging, interface adaptation* and similar duties. This allows application object designers to focus on the domain objectives of their systems and, at a later time, make decisions or experiment with how such objects function in a deployed distributed configuration. Being able to enhance or modify certain runtime configurations is important in dynamic adaptive systems.

More will be discussed about the various services provided by the primary negative space objects, containers and facets in the Design Elements section. For now, we want to introduce the concepts and purpose behind them.

Containers are responsible for "wrapping" all registered objects deployed as a system. The container (not to be confused with EJB containers) will store other negative space objects and special properties at the meta level to the contained application object. Because the application object itself is not required to implement special interfaces or directly interact with negative space objects, the container is capable of providing this capability to ensure that key services necessary for the application to become a well-behaved citizen in a large complex system.



Furthermore, our fractal framework will ensure the appropriate normalization of objects via the use of containers. This permits for implementations dealing with the objectives of dynamic discovery, registration, persistence, and other architecture level concerns to be handled automatically for the application object. Indeed, the application object itself may be a proxy to some specific architecture such as CORBA, EJB, RMI or other where its specific concerns are addressed already. In this case, it is the proxy that is deployed into the fractal architecture and the concerns of the proxy are managed by our framework whereas the remote object is governed by its own implementation.

Other negative space objects called facets can be dynamically attached to application objects or containers. They can perform a variety of useful tasks such as fault-tolerant messaging, dynamic proxy registration, application object serialization, carry digital certificates on behalf of the application object, control access and other similar duties. The main idea behind facets is that their degree of specialization and granularity allow them to be assigned to containers or objects on an *as needed* basis. The framework is able to attach, inspect or remove facets associated with application objects without any effort on the application object itself. In an adaptive system where changing needs drive changes in system functionality, performance or security facets will play an important role.

Fractal Worlds

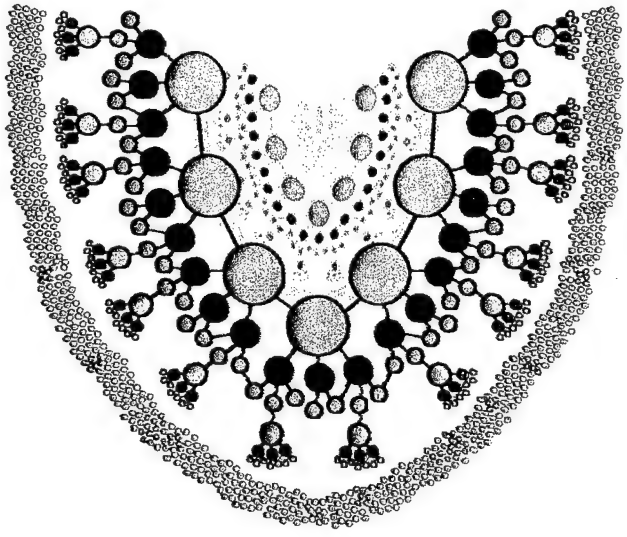
The natural world we live in is truly a fractal one. One of the visions of this research project is to provide a unifying architecture that permits the proliferation of nodes across the internet (or networks) which forms the underlying *terrain* for fractal worlds in the digital realm. Such worlds represent the composition of *systems* and users on many scales in a highly inter-connected composite way. The idea of a fractal-based architecture that acts as a lightweight membrane enveloping each and every "place" (or logical node) on an Internet is a powerful idea that will permit levels of complexity and interaction currently not possible. Using techniques described within this document and others to follow, such large systems can be wielded, connected, disconnected, created, composed, decomposed, adapted, viewed and otherwise understood. This is necessary in order for such systems to arise. Therefore, new techniques for system creation, visualization and understanding will need to be invented, along with advanced tools that implement them.

Communities of Communities

When we talk about building software systems, we can't ignore the fact that there are going to be many users "out there" utilizing different systems for different reasons. The Internet has truly opened our eyes for the need and necessity of connecting users, groups, businesses etc. In the future, such processes will become even more fluid and dynamic and furthermore, the structure underlying these communities will be inherently fractal. That is, *communities of communities* will rise and fall based on temporary or permanent needs. Businesses will be able to create these dynamic communities and deploy them onto an existing distributed topology unified by a lightweight fractal architecture. This organization of embedded population, purpose and structure is not new, and exists in many fields including economics, politics, bureaucracy, the military and others. Realizing this leads us to the notion that such "virtual" communities should bear this same successful organization yet in a purely digital (and fractal) world.

Battlefield InfoSpheres

Based on the vision and design of the future Battlefield InfoSphere (BI) [USAF 1998], we feel that our current work will provide a suitable *landscape* for such InfoSphere environments to exist and proliferate, providing large numbers of connected users with advanced workflow services using mechanisms such as instant messaging, mobile agents, publish subscribe and others. Before such highly connected user communities can interact and share information, an underlying architecture capable of handling and mapping the complex and dynamic nature of these communities must exist. We feel that the characteristics our architecture will provide, namely, the dynamic, adaptive and complexity management features, as well as the fractal scalability are vital to the creation and operation of such InfoSphere communities. We will address this area further in our final technical report.



Miniature & Mobile Device Integration

Because of Java's destination on miniature and mobile devices, it will be possible to connect such devices into an existing system possibly over satellite or other wireless WAN technology. In will become ever important that large distributed communities of users not only be able to access their information from within static structures, but also in very dynamic geographic independent ways. We fully intend devices running Java & Jini to be easily connected into large systems built upon our infrastructure and allow them to attach to existing information channels and receive that information in spontaneous and dynamic ways. Furthermore, it should be possible for such devices to assemble into spontaneous collaborative communities that previously did not exist. For this to happen, common infrastructure technologies like Sun's Jini and our framework (which will make heavy use of Jini) will be necessary to handle implementation and recurring details such as authentication, information routing, and collaboration to name a few. The goal is to handle many of the technicalities involved in inter-object and therefore inter-device communication so that application designers can easily deploy onto any size system or device knowing such technicalities are implemented in a robust way.

Compartmentalized and Secure Communities

When the ability to assemble large communities of systems, users or devices exist, so too must methods for separating and protecting such communities. If such communities are driven by a common low level framework infrastructure, then such an infrastructure should also provide comprehensive ways to secure and compartmentalize systems. It would be wasteful to force system architects to address these recurring concerns in many different ways between themselves when well known and robust ways exist already. Methods for access control, authentication, digital signing, certificate exchange and encryption should be provided by the infrastructure so that designers can easily select the properties needed by their system.

Community designers should be allowed to dynamically create isolation levels that will separate the flow of information between groups of users and groups of groups of users etc. etc. This ability should not reside entirely at the programming language level and therefore the negative space should offer extensive ways to alter the security properties surrounding a community of objects, users, applications, devices and so on.

Technical Findings & Approach

Through the discussions in this section, we hope to illuminate our intended approach to addressing the concerns of this effort as a result of our research to date. In doing this, we will draw from prior efforts where applicable and include our new discoveries and perspectives as well.

The following subsections address the major concerns of the framework architecture; namely, prevalent concepts in its general design, how it deals with technical hurdles uncovered by this research and its fundamental organization.

The following sections appear in order:

- A Distributed Component Architecture
- Event Communication
- Composite Frameworks
- Fluid Architectures
- Adaptable Software
- Federated Object Registry
- Fractal Architecture

A Distributed Component Architecture

The term *component architecture* has become quite popular recently and is bolstered more so by the rise and hype surrounding components in general. However, the term *component* does not necessarily have a commonly accepted definition. As such, we will identify a component in this manner:

> **Definition 2:** *"A component is a reusable element of unspecified granularity, be it an object such as a JavaBean or a larger tier-based implementation structure such as a framework or process that is governed by a well-defined interface."*

Regardless of the apparent size of a component, its fundamental nature within our architecture will be the same, namely it is wielded as any other object with attributes and behaviors.

Our focus throughout this feasibility effort has centered on the Java 2 platform as it provides the most comprehensive set of capability in terms of networking, security, code mobility, and others. The component model specified by JavaBeans thus deserves additional attention in our efforts. It is the goal of this specification to allow for the creation and connection of dynamic Java objects into larger reusable constructs. JavaBeans permit the construction of graphical applications through visual development tools, and we feel that much discovery will be made combining this idea with large distributed systems.

Our architecture will permit for the dynamic deployment and manipulation of components such as JavaBeans (or larger components) over networks thus providing the basis for a distributed component architecture. Objects within this architecture can be just as easily viewed and manipulated locally (with visual tools) as remotely; that is, there is a logical/ physical separation between the organizational concerns of the system and that of the hardware or network. This degree of transparency is important and we talk more about it in the section "*Logical/Physical Separation*".

A component architecture is powerful because it allows for many components to co-exist as a system and interact through well-known interfaces. This allows any component that adheres to such an interface to take the place of another if necessary. This concept directly contributes to the longevity of a system and ultimate adaptability as it allows for new or better components to be included or substituted over time. When you combine this with the ability to organize and/or compose objects into larger grained components and manipulate connections between components dynamically at runtime, you have levels of control over a system otherwise not possible. Furthermore, our system will also provide ways to adapt interface boundaries and include new ones dynamically as well.

Event Communication

Object components within our distributed architecture paradigm will communicate via event generation and delivery. The JavaBean event model provides a clean, dynamic process for components to register interest in one another and receive event class objects that deliver specific information about the occurrence of events within the system. This simple event/listener model allows for the dynamic arrival and departure of listener interfaces on running components in a system; whether local or remote. This quality is essential when constructing, adapting or otherwise modifying systems at runtime. The separation of event generator and event listener via interface allows for components to be systematically linked together in either a design or runtime environment and permits the future inclusion of components (implementing such an interface) into an existing system [Govoni 1999].

Events & Messaging

The manner in which components are linked together via event listener interfaces maps directly onto the object classes themselves. This facility is provided at the class level via an objects event methods as mandated by the JavaBean event model specification. Typically, objects are linked together in source code via objects that perform the necessary assembly of system components. Special tools called BeanBuilders can perform this linking (visually) at design time. One of our prototype tools (see section on *MetaBuilder*) can perform this dynamically at runtime allowing you to assemble, dissemble or modify component links while a system is running without explicit source level modifications or code.

Furthermore, objects in a system built on our architecture can acquire and send object-level messages and events (method invocations) to other objects distributed across a network directly. Before this can happen, objects must be acquired or otherwise resolved in one of the following ways:

1. Using a constant, simple JNDI (Java Naming & Directory Interface) lookup scheme allowing the manner in which objects are discovered and acquired to change (e.g. CORBA naming service, LDAP, RMI) while the interface through which it happens remains constant.
2. Transparently via a connecting tool like MetaBuilder. The lookup details are embedded in MetaBuilder which become irrelevant once a proxy is obtained and registered in the object registry where other objects can access it in a uniform way, namely, as just another Java object.

The nature and implementation of objects communicating across a network is equally transparent. The negative-space facilities provided by our architecture framework will ensure the dynamic proxying of objects registered in the system. In doing this, pure object-level communication can occur yet the nature of the proxy implementation may vary. For example, when an object discovers another object and makes invocations on it, the target object implementation may be RMI, EJB, CORBA, or a standard vanilla Java class to name a few. The idea here is simply that communication between objects occurs in a standard and uniform way regardless of network location or target implementation.

Message Uniformity

Since we've separated the logical and physical concerns, the event passing and component linking mechanisms should not be aware of the network location of components; that is, it remains transparent. Simply put, when connecting objects and components together, the process and appearance of objects is the same whether they are running in a local or a remote runtime environment and is furthermore irrespective of the target object implementation. Furthermore, whether manipulating individual objects, composite objects or object clusters, the process is *self-similar* and should not vary with respect to object size, network coverage or other scale/topology related characteristics.

Distributed Events

In a distributed system where objects and services are subject to blackout periods or removal, a sophisticated distributed messaging facility is needed. Beyond the reliable (and synchronous) object-to-object messaging previously discussed, we also need reliable asynchronous messaging. When objects, components and services go offline periodically, we need to ensure that messages delivered during these blackout periods arrive when such

components and services become available. Along this line, we want to have control over the timeout and duration certain messages have in a system. These concerns are especially crucial in an architecture that can evolve and change dynamically over time. Since components and services have the ability to disconnect and reconnect in a dynamic and transparent way, this ability must not affect the logical flow of messaging throughout a system, only the delivery times of the messages. Key technologies currently emerging in this area will be adopted to satisfy this requirement, specifically, we are looking into the Distributed Event specification for the Jini networking framework.

Reliable, Distributed & Inter-component Messaging

Developing systems across networks requires careful attention to the realities involved. Most notably, that networks and their interconnected computers are not fault-tolerant. Network connections and computers are prone to failure and blackout periods. Systems developed on an unreliable infrastructure such as this need consider ways to manage fault conditions. At the logical level of a distributed system, components are interconnected and communicate across a network. However transparent this appears to be at one level, it is truly not. Given this observation, it is necessary to provide appropriate facilities to deal with the irregularities of such systems. Regarding the need for message delivery across networks, it is our intention to provide *reliable* inter-component message delivery between components at the logical level. The properties of a given *connection* between components can be easily inspected and modified on-the-fly to provide the necessary messaging semantics. This will allow system architects to witness the effectiveness of inter-component messaging connections and alter properties concerning the delivery mechanics. Such properties might include the immediacy of the delivery, store-and-forward capability when destinations are unavailable, asynchronous/synchronous threading etc.

It is our intention that special service facets performing these functions will reside in the negative space within our architecture and be associated with specific components during the construction process. Components themselves need not be intrinsically programmed using one type of connection or any connections at all. Connections and their properties are part of the creation and adaptation of a system and tools such as MetaBuilder (see section on *MetaBuilder*) will allow developers to create, inspect and modify components and their connections across a network.

Event Channels: *Publish/Subscribe Model*

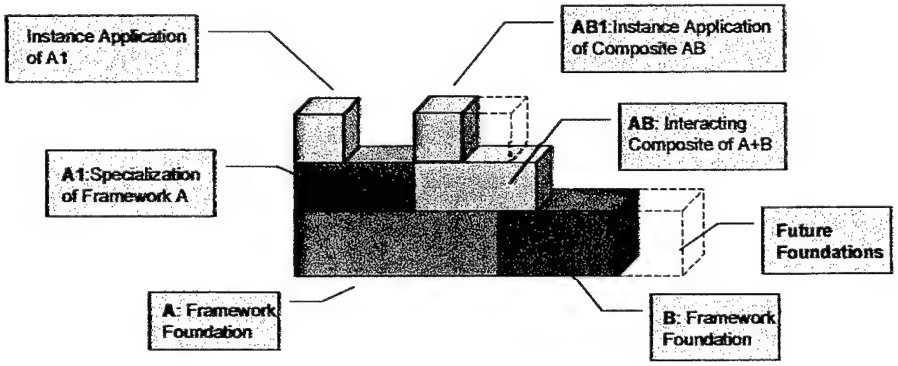
Another interesting form of messaging provided in our architecture is the popular publish/subscribe model [reference here]. In a publish/subscribe system, publishers of data and events are detached from subscribers thereby allowing either to grow or shrink in population irrespective of the other. The most obvious analogy is FM radios. Radio stations broadcast a signal and anyone "tuned" into a specific channel with receive the information broadcasting on that frequency. The signal is not duplicated or re-sent to each individual receiver. This would be too costly and inefficient.

An *event channel* is like a data bus where event objects are broadcast to any and all listeners currently subscribing to the channel. Event objects stream across the channel and provide information about the nature of the event to each subscriber. Events can contain information or other useful behavior driven objects of interest.

Event channels can be secured by requiring subscribers to provide access keys or digital certificates proving they are allowed access to the events and data the channel provides. Furthermore, channels can be restricted based on any criteria a system designer needs such as username, password, group etc. Event channels are governed by ACL's (access control lists) to separate the flow of information among heterogeneous groups of users. The granularity of an ACL is not specified and only restricted by the requirements of the system or application (see section on *Design Elements: Event Channels* for more information).

Composite Frameworks

Composite frameworks are a powerful design technique that allows for the creation of layered frameworks such that implementations unique to a given layer can change independently [Govoni 1999]. Through the use of interacting frameworks stratified in this way, specific technologies mastered by particular frameworks can be embedded, encapsulated and hidden from subsequent layers. This permits for the selective utilization of lower level frameworks and allows the developer to focus more intently on using the top-level or composite framework to solve his/her problem. Composite frameworks are useful and powerful because they provide many technological solutions in one package. The framework designer does not attempt to solve multiple categories of problems through the creation of a single framework, but selects and layers the best framework solutions and provides those solutions through a single high-level framework API. It becomes much easier for the developer to master a single framework with many (possibly unknown) embedded technologies to do various things such as mobile agents, object persistence, object lookups, service lookups etc. Juggling a variety of technologies to do these things would become time consuming and unwieldy. Rather it is the *composite framework* that has brought together these solutions in a concise elegant façade of functionality that permits the layers below to undergo change in either API or implementation without disrupting the applications and systems built upon the composite level framework [Govoni 1999]. It is because these implementations at the lower levels will change over time that makes a higher-level composite framework compelling and useful for system longevity.



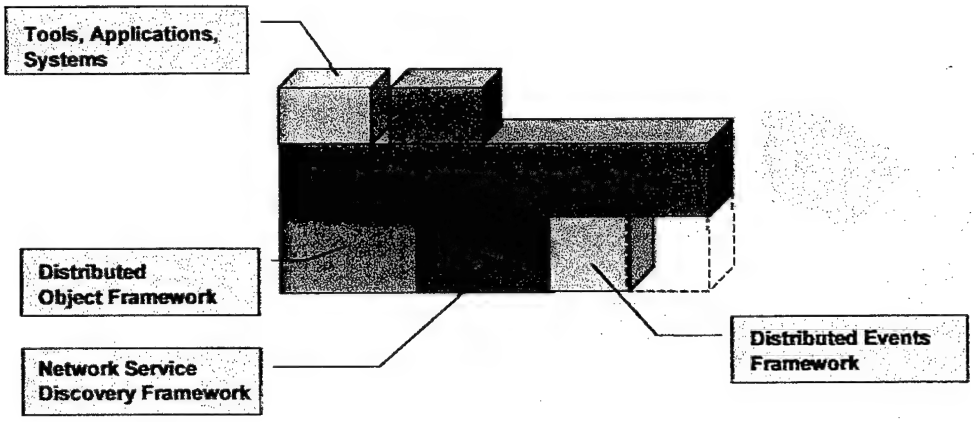
Definition 3: *"A composite framework encapsulates and facades the implementation and interfaces of other substrate frameworks, API's and technologies."*

The lightweight distributed framework we intend to develop and commercialize will likely be a composite framework that utilizes other key framework technologies to solve critical concerns of this research. Some of these concerns addressed by our framework and its underlying technologies include:

- Logical/Physical system separation
- Dynamic object proxying and remote enabling
- Distributed fault-tolerant object registry service
- Dynamic object/application/system persistence and generation
- One-to-many object message/event multicasting
- Fractal composability and scaling

Currently there are available technologies and standards emerging to help make these underlying goals a reality. These include (but are not limited to):

- Java™ 2 Platform
- Jini™
- Voyager™
- iBus™
- XML/BML
- JNDI
- CORBA
- EJB
- RMI

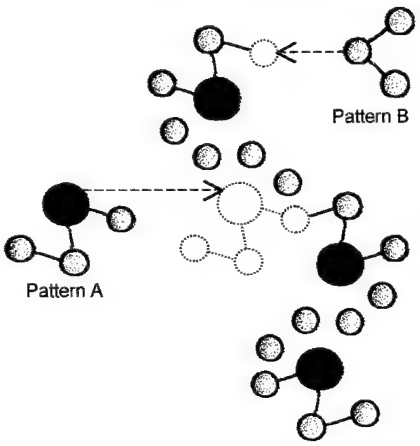


The goal of a composite framework is to encapsulate and hide the details behind the API it exposes. This uniformity gives great power to the developer and also allows the technology components within the framework to evolve independently. This provides greater value to framework users over time; if such technology is provided commercially via third parties, then the advancement of those technologies will likely progress in parallel with other efforts thereby maximizing the advancement of the framework's capabilities and value to developers and users alike. Leveraging commercial efforts and decades of manpower in this manner will not only contribute to the continued growth and proliferation of such commercial markets, but will also benefit software and systems development in the future by providing a stable platform through which such technologies can be developed, exchanged and ultimately used.

Fluid Architectures

When we consider applying adaptive capability to software applications and systems, we create a mental model of fluidity or flowing change. As such change is inevitable, it should be discussed in terms of its smooth migration between milestones. Just as water adapts to the path of least resistance, so too should an adaptable architecture behave like water, that is, form in motion. Today, however, systems are very static and although they operate according to original design and intention, such intentions are constantly moving forward faster then the designs. Therefore, an adaptive software system, be it and application or complex system, in its own right should be fluid and permit itself to be extended, evolved or devolved.

An Architecture of Patterns



In terms of pure architecture or design, software development has made great progress with the identification and practice of design patterns and patterns in general. Design patterns represent micro-architecture solutions to commonly encountered dilemmas in software engineering. Such solutions are truly reusable whenever the problems they address appear and hence are tremendously valuable at quickly solving such problems via well-known solutions.

When we think about architecture at this level, the intent is to build solutions to larger grained problems through decomposition [Govoni 1999], recursively applying well-known solutions to those components. Assembling solutions together into larger solutions is very much inline with the fractal theme of scale and composability offered by our vision.

Erich Gamma notes that one should "*...design a system from the bottom up, applying patterns one after another until you have a system architecture.*" [Gamma, Beck 1999]

This approach to design through decomposition and incremental application of patterned solutions addresses the notions of building complex adaptive systems in the following ways:

1. **Truly complex systems must be incrementally composed to be well understood and managed**. No large system can simply come into being, but rather it must grow into being. As a system grows into being, it's growth should be governed by concise, logical components. In this case, self-contained patterned architectures to discrete problems within a system can be designed and assembled into the system over time thereby evolving its complexity and its very design at a macro-level.
2. **A macro-architecture composed of well-known micro-architecture solutions can be better controlled.** Because the sub-components of such a system are themselves well-known solutions to problems, the overall strategy of the system architecture evolves implicitly rather than explicitly. Such an approach allows the discrete problems contained in such a system to be well understood and solved outside of the greater context, for the greater context is much larger and more difficult to grasp and solve at once.
3. **Progressive development and systematic adaptation means greater productivity.** Not only does decomposition allow for the stepwise creation of large complex systems, but the requirements and demands of such a system will likely evolve over time as well. Addressing these incrementally by incorporating small pieces into a system rather than re-designing or rebuilding a new system on longer schedules will keep users productive as needed functionality is layered in progressively, in a more timely manner.

Inter-Architectures

When we talk about creating small architectures and linking them into a system we are exposing a larger implicit architecture that continues to evolve over time. This process defies initial design attempts to capture all foreseen capabilities of a system into a comprehensive design. And as surely as such capabilities are to arise with time, so too should such a design unfold over time.

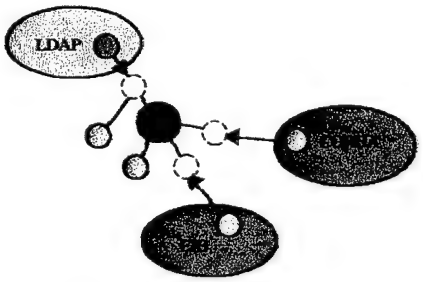
As we build mini-architectures to address functionality and lock them together with pre-

existing ones, we can see that we are creating a network of architectures connected together much like the internet is a collection of inter-connected networks. Therefore, we term an *inter-architecture* to mean the following:

> **Definition 4:** *"An architecture permitting the interconnection of multiple disparate architectures whose design is manifest through such connections."*

Each architecture thus connected could have evolved through it's own design process and requirements. The ultimate super-system emerges as individual systems are collectively joined and continue to undergo evolution. This system in turn can be further linked to other systems to create a magnificently fractal landscape of *systems of systems* joining countless users. In order for this to occur, a common mechanism by which these systems are built and linked must exist. It is our goal to provide the thin membrane that envelopes these systems, that is, an architectural framework for deploying objects, components and applications into a dynamic environment where such interconnections are now possible.

Furthermore, since this infrastructure will reside in the negative space, objects and components can be discovered and deployed irrespective of their particular implementations be it, CORBA, RMI, EJB, or any other type of object that can be discovered or otherwise instantiated as a Java (currently) interface or class.



Different object implementations being unified in a common architecture environment.

Logical/Physical Separation

Current software systems tend to be very rigid in their design and bound to a specific network or hardware level configurations. This is to say that concerns regarding specific computer systems and their locations within a network intrude into application level concerns and often manifest in design and implementation details. This unfortunate historical bond between application (logical) and network (physical) level concerns prohibits flexibility and adaptability at either level; since the two levels are generally woven together in present systems, changes in one often results in severe side-effects in the other.

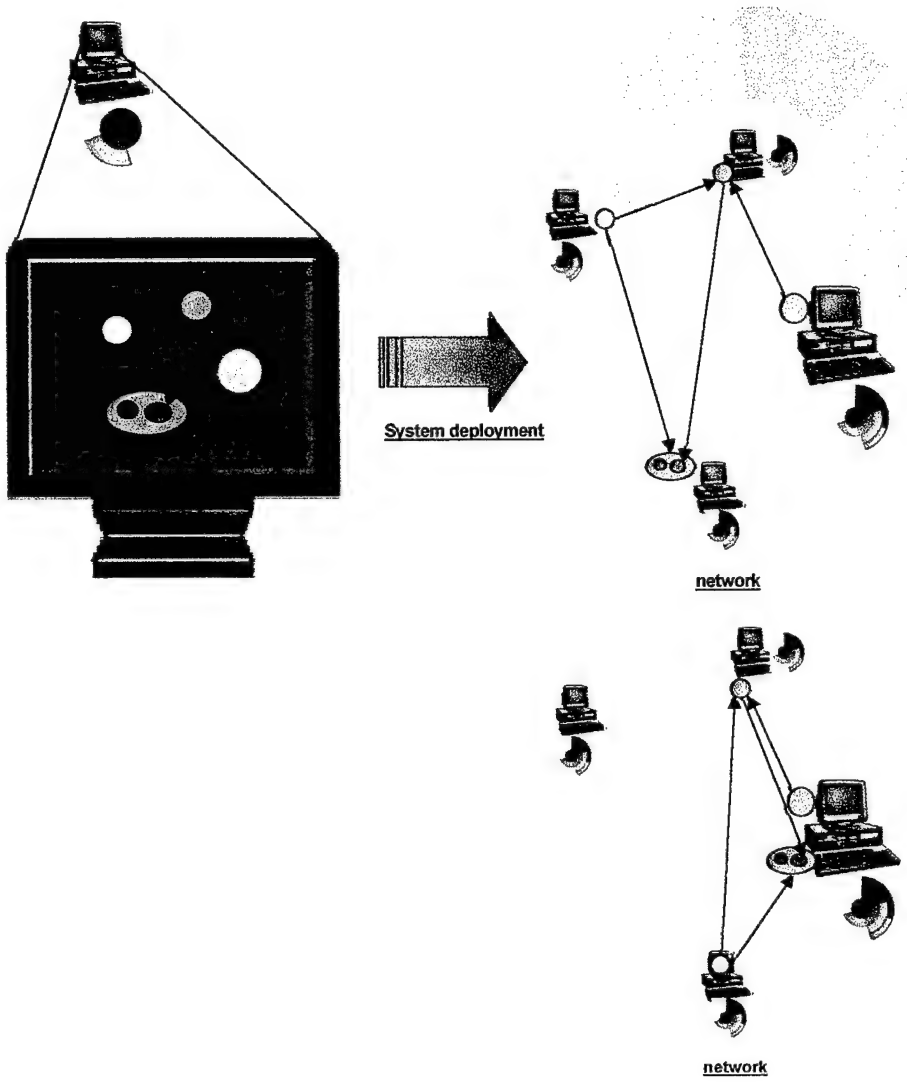
If we consider the architecture of a complex software system independent of any given instance of it, configurations relating to the physical instantiation of the system should not be present. This simply means that a design is bound to specific configurations when it is instantiated. Many such configurations may exist, but in fact, typical software systems are rigid and bound to a specific physical configuration. It is our belief that perhaps the single most important factor in designing adaptive complex systems is the appropriate separation between the logical design or architecture from the physical, network or hardware configuration. Therefore let us present the following corollary to express this.

> **Corollary 5:** *"In order for the logical aspect of a system to evolve, it must not be bound to physical aspects that do not evolve or are not relevant to its logical design. Therefore, some degree of separation should be present between logical and physical aspects of a given system."*

Likewise, let us make the following observation that we feel is central to designing and building complex adaptive systems.

> **Observation 2:** *"With the appropriate separation of logical and physical concerns, either should be able to undergo adaptation or re-configuration without disturbing or adversely affecting the other."*

There are ample precedents for such separation in more mature fields such as database design (logical data modeling vs. physical database design) and VLSI hardware design (logical modeling vs. physical layout).



It is a major theme of the prime objective to be able to construct useful applications and systems from granular components dynamically and intuitively. The fruition of this idea relies on a few basic assumptions:

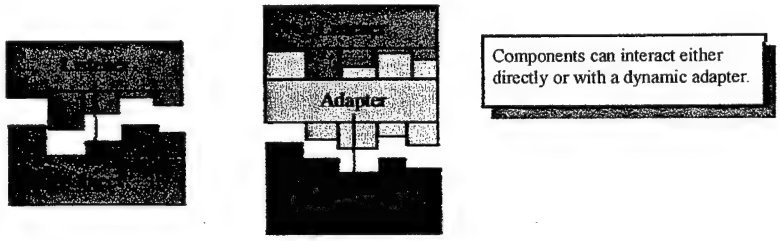
- **Applications should be developed by with reusable components and interfaces.** The old days of developing monolithic applications is over. Today, developers must take the approach of designing reusable components and using or assembling those components into their desired applications. Whether this happens statically or dynamically isn't as important as the approach itself. Our objective is to allow these components to be mixed and matched between applications to create new hybrid apps that better serve a users dynamic needs.

- **The market for macro and micro components of varying granularity will rise.** Once the use and design approach of component applications and architectures proliferates, markets for such reusable objects will also emerge as it is now doing. Tools and paradigms taking advantage of this trend will be at the forefront of a new age of software design & development (witness IBM's alphaWorks site for beans and visual building: www.alphaworks.ibm.com).

Interlocking Components

It is through this paradigm of granular interlocking components that adaptability can be obtained. Individual objects can be linked into applications at runtime and connected to specific events. In doing this, an application can have new functionality added to it by connecting it to new object components, possibly acquired from other applications.

The manner by which objects can be connected is already present in the JavaBean event model and is based on the clean separation of event generators and listeners via simple interfaces. Building upon this foundation, adapter objects can be generated dynamically by introspection tools like MetaBuilder to connect two components that do not share a common interface. With this ability, any number objects can be connected and ultimately interact with each other.



Observation 3: *""In an adaptive architecture, it is important that components are allowed to change, be added or removed in addition to the connections between them" [Oreizy et. al. 1999].*

Interacting Applications

We would further pursue to extend this useful paradigm across networks where components residing on different hosts can interact and be visually connected in much the same way.

One interesting aspect of all of this is the potential for different applications to interact with each other. That is, functionality from one application can trigger or be linked to functionality of another and share information as well as processing capabilities. Furthermore, in a dynamic networked environment, this ability will promote the construction of large dynamic distributed systems where specific functionality is geographically disperse rather than redundantly installed on every host or client machine.

We envision advanced tools like MetaBuilder that will inspect applications and systems and allow objects to be connected to events of other objects regardless of their context, be

it another application on another host. With this ability, creative new distributed workflows can be conceived and implemented in a fraction of the time it would take to design a entirely new system or modify an existing systems source code to achieve the same results.

In an earlier section, we talked about long-term and short-term adaptability. In the future, we see specific needs arising very quickly for short immediate periods. If an extensive library of components are on hand with which to construct usable and functional data views or mini applications to meet those needs, then a mechanism to do so would certainly be valuable and reduce the costly and lengthy cycles of folding short-term needs into large long term systems and applications (see section on *MetaBuilder*). We use the term *transient application* to mean the following:

> **Definition 5:** *"An application whose purpose, lifetime and construction respond to short-term, immediate or otherwise non-permanent external needs."*

The tools and technologies Metadapt intends to develop will pave the way for end-users to quickly link, assemble or re-arrange useful components to accommodate time-critical demands that may be more immediate and short-lived and thus opening the possibility to creating tools and applications with similar short-lived utility.

End-user programming provides a way for end users to enhance, modify or even create applications suitable to their specific needs or those of their team or group. The inefficiencies of developing one-solution-fits-all software for large diverse business are becoming well known. We believe that mechanisms for enhancing, adapting and creating mini-tools and applications specific to individual users, tasks, demands, or other objectives any of which might be short term and immediate in nature.

Part of our vision revolving around the prime objective of this research is to provide a gateway into the adaptive, creative facilities we will embed within our framework. This gateway may take many forms such as an IDE for developers, an embedded builder tool (similar to Visual Basic Editor for Applications) and other usable and embeddable technologies. The ultimate goal is to provide pervasive, easy, visual metaphors for assembling, components, applications and systems. Empowering internet/intranet users of tomorrow will involve giving them the ability to create specialized tools and connected different applications together in interesting purposeful ways. This is the goal of end-user programming as we see it.

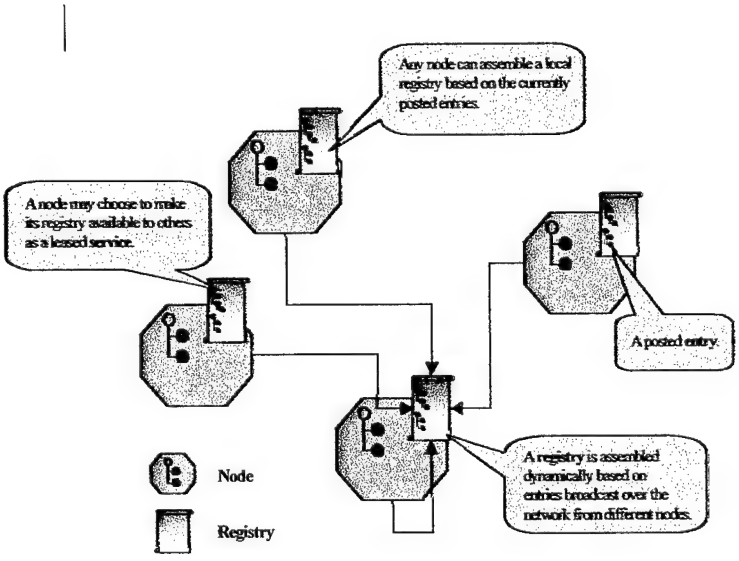
Federated Object Registry

Within a distributed object-based system, it is important that a uniform mechanism for searching and acquiring objects exists. Because the details about where components might be located and whether, in fact, they are available may be unknown at any given time by a lookup client, it is essential that a uniform entry-point into a system exist. Furthermore, it may also be unknown where the lookup registry is physically located *and* whether it too is even available. For these reasons, we intend to develop a location independent *federated object registry* through which objects can make themselves remotely available to other objects and applications between and within networks.

The Registry Federation

In this approach, there exists no centrally located registry process that tracks the whereabouts of objects in a given system. Rather, every *node* within the system maintains its *own* local registry. Objects within a local node post their registry entries to the only known registry, the locally running one. All nodes in a system operate in the same manner, and thus collectively, all registries represent the whole of every registered object within the system. Using multicast protocols, clients wishing to find the location of individual objects, components or applications can simultaneously see all federated registries within the local network. Given this instant combination of all active registries, an object may query for other objects by name, type or directory across the entire network.

This approach maintains a high degree of network and registry fault-tolerance. If any node or registry becomes unavailable within the system, that portion of the federated registry is simply not seen and therefore not search. In order for there to be no registry service available, there would have to be no nodes, implying that no system is running.

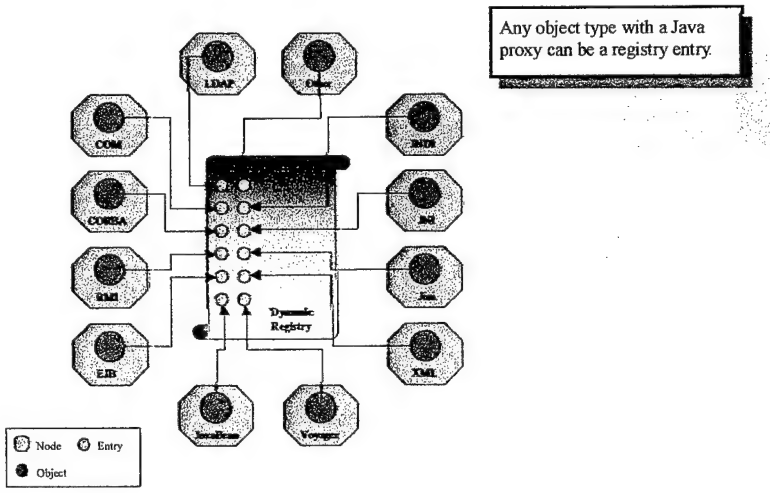


Although the distributed aspect of the federated registry may cause a decrease in lookup performance; however, one can easily implement a central optimized or cached registry that simply acquires information from the federation and makes it available through a well known or posted location to clients in those places where lookup performance is a concern. Further investigation into these tradeoffs will be performed.

Object Entries

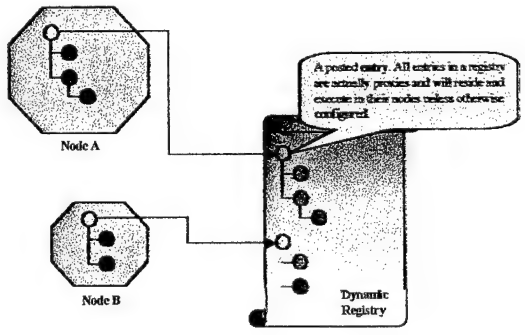
Within a given registry, objects can register themselves both by name and location. Specifically, a registry is much like a file system with directory structure. Objects can add themselves to an existing registry directory structure, or create their own. This allows clients to look for individual objects by name or many objects grouped into a common directory. Furthermore, objects registering with their local registry need not add themselves directly to the registry, but rather can have a remote proxy added on their behalf. Because other objects across the network will discover and use this object as a service, it must be able to communicate across the network. Any Java proxy

implementation can be added to a registry; if the object is not already remote enabled, a remote enabled proxy will be generated for it dynamically.



When a node chooses to register a remote proxy for an object, all discoveries made on that object receive the remote proxy and not the object. In this case, the actual object itself never leaves the node it is running in; rather, it is accessed remotely across the network. Alternatively, an object may choose to register a serialized copy of itself in the registry, which may then be downloaded and used by the client directly, eliminating the need for subsequent remote communications. Clients need not be aware of the location of objects found through the registry.

The following diagram shows two nodes registering object trees in the same registry. Even though each node has the potential to publish its own registry, this need not be a restriction.



Dynamic Discovery & Joining

When a registry comes online, it announces its immediate availability via IP multicast (specifically using Sun Microsystems Jini™ Technology). When this occurs, all interested nodes, objects or applications are notified and can respond accordingly. At this point, when clients access the federated registry, the newly available registry component will now be a part of any queries or lookups performed by clients.

Since these processes are detached and truly dynamic, the ability exists for large systems to undergo casual attrition of components and registries without breaking the entire system.

A Fault-Tolerant Service

As we noted, this approach to a network registry affords very complex systems a high degree of availability. This is extremely important in dynamic complex adaptive systems where components, subsystems or portions thereof undergo evolution, such as the addition or removal of components in a runtime environment.

The ability for a system to recognize the presence or absence of key components and make necessary adjustments is vital for its proper and continued operation. The following observation becomes evident.

> **Observation 4:** *"In highly dynamic and distributed systems recovery from partial failures is critical to the continued success and operation of such systems."*

Therefore, an appropriate mechanism must exist at the architecture and software level to account for these realities.

The One Registry

A given population of registries can be as large and ubiquitous as a system architect sees fit. There can be a single centrally known registry if applications are developed with such knowledge; however, the unity of a federation of registries is clearly a powerful mechanism to build large interconnected systems.

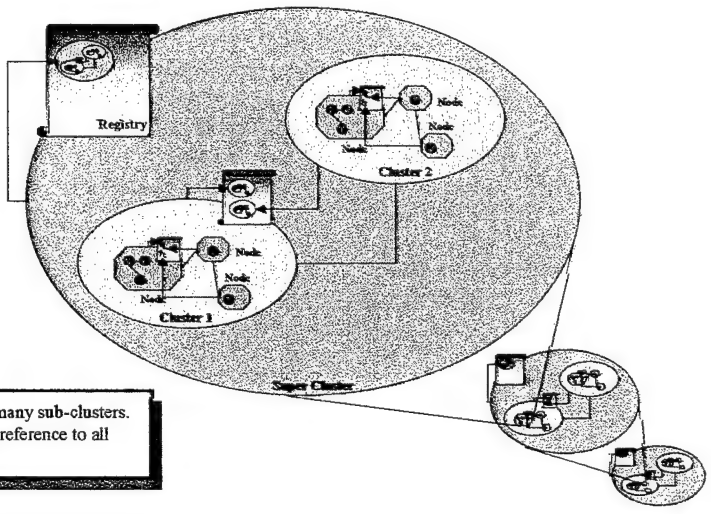
The perception to client applications remains relatively simple. To the client performing a lookup, it appears as though there is a single registry somewhere on the local network through which objects can be located and acquired. For this reason, this meta-level notion of a single registry is referred to as the *One Registry*. Rather than being an explicitly maintained process on a specific physical computer, it is manifest through the unity of the federation.

> *The One Registry is manifest through a constant interface which the client uses to perform queries and operations. The implementation of this interface manages the communications between the registry component federation dynamically.*

The *One Registry* can be as large or as small as required and can grow quite boundlessly, providing structure, services and emergent organization from the collection of nodes and subsystems underlying it. Although, the precise number of nodes, their locations, duties and purposes may change based on human needs and demands, the existence of the *One Registry* is constant, reliable and reactive. Also, the fundamental notion of viewing a "many" or multitude as a single or a "one" is pervasive throughout our design approach and is discussed more in *Fractal Architecture: The One and the Many.*

Registry & Cluster Composition

For each node in a distributed complex system, there exists a registry responsible for the entries held in that node (or cluster). As noted earlier, collectively, all registries combine to form the One Registry acting as a single visible entity. Since registries, nodes and clusters can themselves be entries in a registry, this allows for interesting scalable compositions of clusters and systems. Furthermore, through designated entry points, large hierarchical systems can be easily navigated through nested registry entries. Such systems may well be separated by vast distances, sub networks or security restrictions, however, given the appropriate authorization of the client, navigation can be easily accomplished.



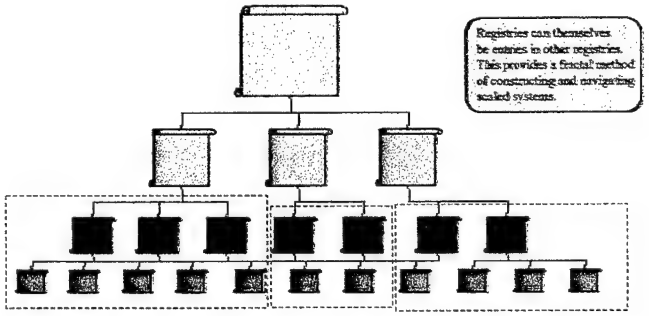
A super-cluster composed of many sub-clusters. Each *one registry* maintains a reference to all clusters visible at that level.

Hierarchical Structure

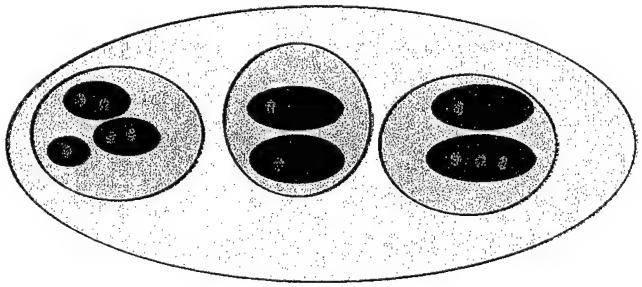
The ability for registries, nodes and clusters to themselves appear as entries in other registries allows for large systems to be hierarchically composed and connected. A few reasons why this is useful might be:

1) The need to mirror the collective organization of human structures outside the system.
2) The ability to traverse all connected systems.
3) The ability to acquire and share applications and data across systems dynamically.
4) The need to connect and create communities of users for collaborative, organizational or social reasons.

The same systems may themselves be sub-systems appearing in different super-systems. This method allows for the isolation of concerns and dynamic integration of systems at the upper levels, which can occur without intervention of (or changes in) lower level systems. The following diagram shows a color coordinated visualization of a registry super structure in relation to system integration and containment.



The following containment chart represents the clusters whose registry structure is depicted in the above diagram. The colors of the registries in the previous diagram match the clusters they belong to in the following diagram.



Using the registry structure, an automated process or system object can easily decompose the structure of the super system and/or discover important objects or information at any particular level.

This facility is highly applicable in an agent environment where agents need a way to travel through specific routes in and between systems and make useful discoveries along the way without the benefit of knowing precisely where registries or data objects are actually located or if they are available.

A Colony of Registries

The resilience of the *One Registry* is quite akin to animal or insect colonies in that there exists no central authority for controlling operations within it (unlike the central reproductive capability of most insect colonies). The benefit of this type of colonial organization in many species is integral to their survival. Colonies have the ability to adapt quickly and can undergo waves of attrition without disrupting the higher-level purpose of the species or complex system (see section on *Natural Systems: Ant Colonies*). In tandem with our objective, it is essential that complex adaptive systems built with our framework components be able to exhibit high levels of robustness and resilience as well.

Fractal Architecture

A *fractal architecture* is one that exhibits fractaline qualities or is self-similar, demonstrating an invariance in form, structure, or function with respect to scale. In the section on Fractal Design, we covered some of the basic notions behind fractal sets and their geometric origins. We also presented some interesting terminology issues between the worlds of fractal geometry and software. Namely, the use of *invariance* and *scale* should be accompanied by appropriate semantics when used in the context of software architectures and certainly this issue warrants further research.

One of our goals is to discover a way to reproduce the visual geometry offered by classical fractals in the more abstract and geometrically void world of software and object modeling. By doing this, we create a natural manner with which to build and understand complex fractal architectures. Without an appropriate visual isomorphism decomposing and understanding the design and complexity of such a system is seriously diminished.

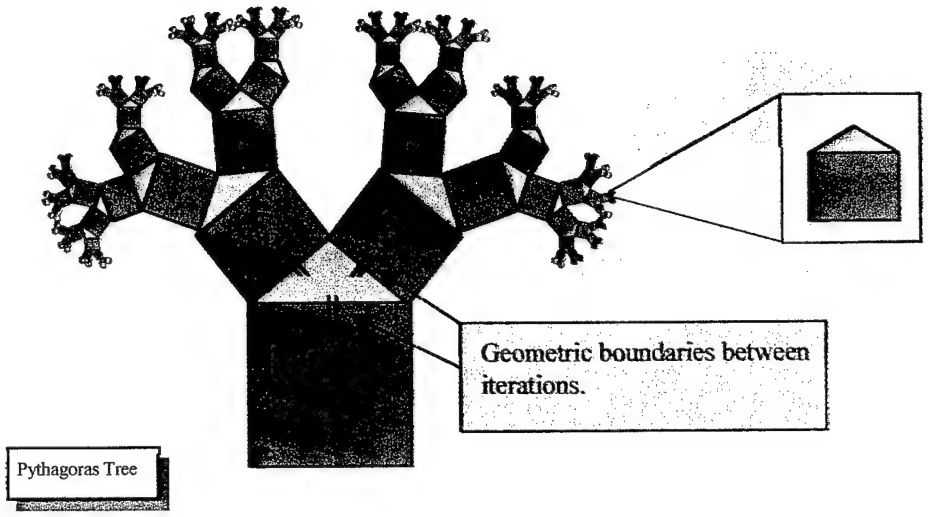
Abstractions Without Form

The qualities provided by classical fractals are geometric in nature. That is, the self-similarity is that of *form*. With different forms come different geometries and different images we come to know as fractal sets some of which include the popular Mandlebrot and Julia. Any application of the term fractal including it's classical meaning to another discipline outside of geometry will require a close examination of how the properties of form can be manifest and subsequently interpreted as such. For to claim that such an abstraction is fractal in its nature is not enough. Such fractaline qualities should be manifest in form and function and easily interpreted as such. Without the understanding of this self-similar form, the benefit to human perception and understanding is diminished. The problem with such qualities in an object-oriented world driven by an object-oriented programming language is that objects, in their most unqualified and pure representation are basically, without form. That is, they contain no extractable qualities of shape or symmetry that could easily be mapped in a two or three-dimensional environment. Since we are concerned primarily with architecture, we cannot assume the object implementations will provide the necessary form bearing properties through which a reliable quality of fractal can be perceived, rather it may be that the composition of objects into designs, patterns and architectures manifests a self-similar repetition of organization and process. We will explore this further throughout this section. Let's continue by examining the relationship between the visualization and representation of fractals and fractal structure.

Mapping Boundaries & Form

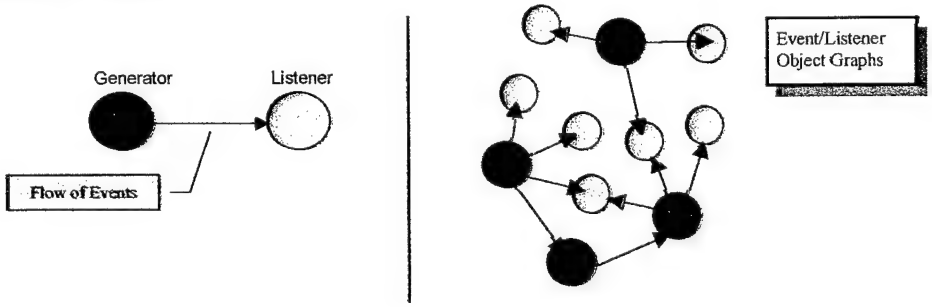
In IFS-based fractal sets, there are typically noticeable boundaries between iterations that are affine transformations of the same structure. These boundaries mark the geometric attachments for future iterations of the fractal. Specifically, as the fractal generating process iterates, scaled and/or transformed versions of the fundamental structure are associated with the super-structure through geometric proximity. [see Pythagoras Tree]. The exact placement of generations of shapes into the fractal structure is driven exclusively by geometry (which is to say the generating procedure). Therefore, the overall structure is governed by the position, scale and transformation of fractal shapes on these boundaries. Such boundaries are mathematically represented and easily interpreted visually. In order for a high-level fractal super-structure to exist, it must contain micro versions of itself interconnected through explicit symmetrical boundaries. Depending on the complexity of the fractal, these boundaries may be easily seen. Certainly, the orientation of such boundaries contributes to the overall form of the fractal set.

To speak of an abstraction bearing *fractaline* qualities, is to say that such qualities are represented in some fashion and therefore subject to visualization. Such a visualization should maintain proper isomorphism with the underlying representation. Any such visualization should manifest in geometric ways; whether 2D or 3D is not quite as relevant. Given these observations, an appropriate mapping between geometric form and boundary need be in place for any such isomorphic views of a fractaline abstraction to exist.



It is our belief that such mappings can exist, and that through advanced tools utilizing and manipulating object properties in the negative space, fractal forms can possibly be mapped onto complex object graphs. Although, maybe inferior to the beautiful variety we see in classical fractal sets and nature, structured self-similar designs can be created, deployed and controlled. Furthermore, additional form bearing characteristics may be extracted directly from running systems through state decomposition [see section on "*Describing Complexity*"]

The possibilities for form-bearing properties may include object connections (e.g. event generators & listeners), embedded object groups or clusters (i.e. containment), dynamic event adapters, logical spaces/domains and others we are exploring. For these sorts of entities, there exists some fundamental visual representation that is true to the abstraction. For example, consider two objects where one generates an event and the other listens or receives the event. Quite basically, such a relationship can be visualized much like in the figure below. Here, the two objects are represented along with a connecting line indicating the flow of events.



Using this simple representation, graphs describing any and all object event/listener relationships can accurately be visualized. Such graphs have been referred to as *attributed graph grammars* and as such they

"*...can represent the set of all an application's possible configurations where architectural changes are regarded as graph-rewrite operations.*" [Oreizy et. al. 1999].

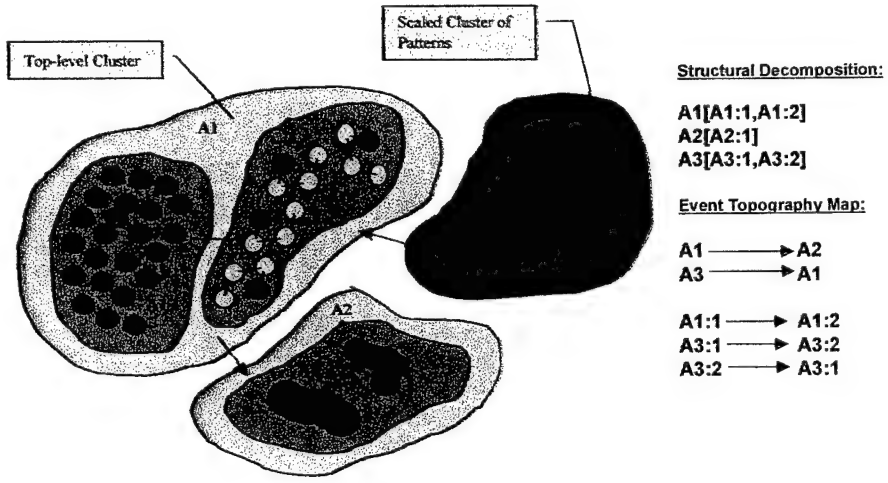
Therefore, it is our goal to isolate form bearing relationships among objects that can be tracked by generating tools (see *MetaBuilder*) and recorded in the *negative space* on a per-object basis such that traversing all objects in a system (i.e. it's graph) can provide the necessary information needed to visualize the fractline properties. Furthermore, such properties should demonstrate invariance with regards to scale or magnitude.

Organizational Constructs & Clusters

Within our architectural framework, lightweight containment objects called *clusters* will provide dynamic object grouping and organization. Specifically, heterogeneous (or homogenous) groups of objects and *micro-architectures* (patterns) can be organized into higher level constructs. Clusters can furthermore be grouped into other clusters and subsequently linked together similar to individual objects, sending and receiving events in the normal fashion. The object cluster is a logical construct and is not bound to physical properties of a system; for example, an object cluster may span many nodes and therefore many processors/computers. Members of an object cluster (including nodes) can be added and removed dynamically as well as belong to more than one cluster. Furthermore, clusters will have the ability to automatically scale themselves by increasing or decreasing the number of members within the cluster accordingly (and possibly contain heuristics for load balancing and distribution).

Object clusters can be assembled, deployed and attached into existing systems dynamically. With the aid of visualization tools, clusters can be observed and manipulated by a system architect; they may also be manipulated by non-human entities such as agents or processes.

The inherent recursiveness of clusters provides a form of simple self-similarity that bears a quality of fractal. The *cluster* is an object that addresses containment and containment is a fundamental form-bearing characteristic intrinsic to the notion of object composition and OOP [see Design Elements section on Cluster for a more detailed design overview].



Describing Form & Structure Using Grammars

It is our intent to explore ways to describe systems constructed using our architectural elements and framework. An accurate description of the architecture and organization of a system should be extractable without requiring any additional functionality to be implemented by the application objects themselves (such functionality will be provided through the negative space). Since such systems will be built fractally, an appropriate self-referential (i.e.recursive) grammar for describing such structures would be useful not only in capturing an existing architecture, but for generating them as well [see section on *Generating Systems & IFS's*].

In the prior two sections, we talked about the need to represent form-bearing qualities of an otherwise abstract system such that an isomorphic visualization can occur. At present, the fundamental notion of *containment* provides the only foreseeable intrinsic property of objects that can be assumed present. Beyond that, form properties within the negative space will need to provide meaningful information regarding the logical structure of a system. More on describing form & structure is presented in the section *Representation and Codification Schemes.*

Given the nature of object oriented systems, the following classification seems to result:

- OBJECTS
- GROUPS OF OBJECTS
- GROUPS OF GROUPS

Containment is provided *ad infinitum* through the group, or what we refer to as the *cluster*. This classification provides a boundless degree of scale through which objects of differing granularity can be composed, organized and/or linked in a uniform manner.

If we wanted to formalize this simple classification into a BNF (Backus-Naur Format) grammar, we might end up with something like this:

```
META-CLUSTER ::= (CLUSTER*)      // 0 or more clusters contained
CLUSTER ::=  (OBJECT*)           // 0 or more objects contained
OBJECT ::= object                // 1 object
```

The cluster of clusters, or *meta-cluster*, provides a self-similar method through which groups of homogenous or heterogeneous objects and architectures can be logically organized independent of their physical location or low level design composition. Logical structures like *clusters* can be imposed on an otherwise flat, complex system and provide a means of navigating and partitioning discretely functioning subsystems within a larger system. The presence of levels such as this will permit the *incremental* viewing or scaling of systems composed of millions of objects whose ultimate architecture may contain many embedded levels of concern that are logically composed [see section on *Manipulating Different Levels*]. Furthermore, such *organizational* constructs do not intrude on the stratified architecture of the objects themselves, allowing them to undergo mutation or evolution without disrupting the discrete levels of a complex system. This separation between layers and levels will make such systems easier to navigate, construct and adapt.

Attributed Graph Grammars

Attributed Graph Grammars (sometimes referred to as *attribute grammars*) are commonly used in compiler design, natural language processing and other forms of structured parsing. Essentially, an attribute grammar (AG) is a graph or tree representing the combined grammatical structure of a given input string (or some parsable structure) using hierarchical or inheritance models for determining node properties. The fundamental properties of nodes within an AG are based on the rules of the grammar relative to their contextual location. For areas such as language, different properties will result for a given node depending on its immediate relation to nearby terms. The rules governing the application of properties for a given context are part of the actual *grammar* being represented.

The notion of applying AG to an architectural design implies that some structured set of principles or rules can be generated and used within the grammar parser to represent the structure of a design (vs. say a language). Given an appropriate set of design rules and/or properties, a set of entities (possibly objects) can be traversed and their subsequent context parsed. This would result in the generation of an appropriate AG where each node represents a corresponding entity or object (whereas in language it would be a word or term). Likewise, the properties associated with each object node would result from various intrinsic and extrinsic design properties. These might include the properties of the object itself, its clusteral rank as well as its various connections and links to other objects.

Depending on the chosen design properties used to represent an architectural grammar, there may be many attribute graphs that one can view for a given system, possibly giving them insight into different aspects of it. Furthermore, assuming the adaptive nature of such a system, new attribute grammars will likely arise and their visualization would be important in understanding the evolved nature of the system.

The One and the Many

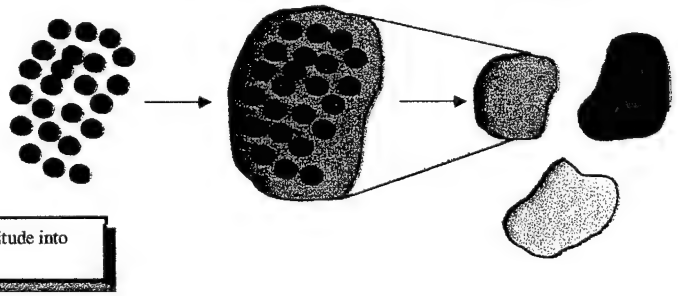
Dating back to the days of Plato, the existential questions of oneness and multitude have been described as the *one and the many problem* [Rucker 95]. To philosophers and set theorists the issue is deciding what constitutes a *one* or a single thing. Certainly all of the universe is based on composition; that is, every *thing* seems to be composed of yet other *things*. So in one sense, every thing is actually a multitude or a *many*. However, we certainly do not perceive or even think about these *manys* that way. Rather, our minds tend to create very distinct and discrete symbols for single things. This arises out of the need to chunk information together [Hofstadter 79] into more easily storable and graspable concepts that can not only be thought about reasonably, but also discussed in a reasonable amount of time. Without this ability to consider multitudes as single entities, we'd be caught in an endless cascade of semantic decomposition when thinking about even the simplest of matters.

Indeed, the link between our minds and our perceptions has quite nicely automated this task and our intuition leads us to accept this as a natural means of understanding concepts. As concepts build upon prior concepts, we find ourselves acquiring ever more complex sets of information, yet we are still able to converse about such things without undue effort. This is precisely how humans learn.

Therefore, a critical capability of our fractal architecture is to allow higher order structures of multitudes to be perceived and manipulated as single entities, or, in other words, treating a *many* as a *one*.

This serves a number of important objectives of this research:

1. **Encapsulating *multitude* into *oneness* decreases the amount of information and therefore complexity**. The complexity of multitude is artificially reduced by managing the *amount* of information an observer must process. If the complexities within a system can be chunked into larger grained entities, then the perceived complexity is reduced. The actual complexity is unaffected; it is simply not observed simultaneously.
2. **Human efficiency benefits from the ability to wield large multitudes as small, single entities**. As single entities are more readily stored and retrieved, the more information we chunk in our memory, the more efficient our storage and retrieval mechanisms become.
3. **Human perception benefits from maintaining a constant amount of visual information for which to digest**. Because of our natural limits to gathering, processing and storing information, we must take a piecemeal approach to understanding complex systems. By imposing boundaries to subdivide and organize multitudes, the inherent structure of complexity can better be controlled, designed and understood.
4. **Embedded composition of oneness is a recursive self-similar mechanism of creating large complex entities**. Empirically, the world around us is simply one giant organization of interacting elements, ranging in size from planets to microbes. The boundaries between single entities and multitudes or systems is more or less subjective on the part of the observer. It is apparent that the larger systems are composed of smaller systems and this is mostly a self-evident observation. In the spirit of this observation, artificial entities in a software system will organize similarly. By providing the constructs that impose organization boundaries on multitudes, they can be organized or grouped into single system entities. Such single entities can further be composed into yet larger entities and so on, ad infinitum. This is a recursive, self-similar ability with an inherent quality of fractal.



Clustering multitude into Oneness

Human Perception & Understanding

Drawing upon these observations, a goal of our research is to create a stronger tie between human perception/understanding and the creation and existence of massively complex systems. In order for these systems to be understood, measures not too distant from our own "hard-wiring" should be employed where possible. Within all of this are very subtle, yet reoccurring qualities of fractal. For example, the composition of concepts and knowledge.

Any such fractal architecture should at some level manifest in ways consistent with the known facts of human perception and understanding as well as the way the universe is designed.

To be truly fractal is to be like nature. Nature relies heavily on composition and self-similar organization. It is the most efficient form of *representation* and this would be why it is so prevalent in nature. Therefore, our synthetic representation will be of the purest form of self-similarity and structure not too different from the compositions in nature.

Universal Set Theory

Universal set theory attempts to provide meaningful statements about the categorization of groups (of elements), which is to say *sets* or collections. A set is simply the notion of multiplicity grouped together. It is this grouping of multitude that creates a set. Set theorists pose that an infinite number of sets exists and it is only a matter of how you choose to perceive them; however, we'll be well grounded in the finite.

The basic operation behind the creation of a set is inclusion. Sets contain elements and elements are considered to *belong* to one or more sets. For example, let A be the set of all ants. The sentence 'χ has wings' is true for some elements of A and false for others. To indicate this new subset we would write:

$\{ \chi \in A: \chi$ *has wings* $\}$.

Which says that χ is an element of A and χ *has wings*. The statement defines the boundary or requirements governing inclusion in the new subset. Likewise, we also have:

$\{ \chi \in A: \chi$ *does not have wings* $\}$.

Therefore, it would seem, that A actually contains at least two subsets: call them B, C (*axiom of pairing*).

B: $\{ \chi \in A: \chi$ *has wings* $\}$.
C: $\{ \chi \in A: \chi$ *does not have wings* $\}$.

The statement ($\chi \in A$) tells us that χ is a member of A. The sentences (χ *has wings* , χ *does not have wings*) provide the parameters for χ to be a member of B and C respectively. Such sentences should provide a boolean operation that produces either a true of false result indicating whether a given element is a member of a set or not.

Set theory gives us the tools to express relationships between sets and their elements [Halmos 60]. It also defines a number of axioms that provide the rules governing sets, how they can be formed and their known characteristics. We will not discuss the axioms here, but merely reference them when they apply to our discussion.

Elements can belong to more than one set and for this reason, it is useful to talk about derived sets that represent the *unions* and *intersections* between other sets. To say that χ is a member of either set A or B or both we would write:

$\{\chi: \chi \in A$ or $\chi \in B\}$

Which says that χ belongs to the union of A and B if and only if χ belongs to either A or B or both; it follows then that

$A \cup B = \{\chi : \chi \in A \text{ or } \chi \in B\}$

which can also be expressed universally as: $\mathbf{U}\{\chi : \chi \in \{A, B\}\}$

The intersection between A and B is similarly expressed as
$A \cap B = \{\chi \in A : \chi \in B\}$, meaning that χ must belong to both A and B.

The intent of our fractal architectural framework is to provide a degree of uniformity across scale of composition, which is to say from designing in-the-small to designing in-the-large. This process focuses on composition of elements (objects) into larger scaled entities, (even across a network).

To put our discussion of set theory into context, consider the objects comprising a given design, irrespective of scale. It might be a composite object made up of other independent objects, a pattern or mini-architecture, or a large complex architecture or system. Objects within these constructs can appear throughout a given system, possibly being re-instantiated or acting as a service. In the case of service objects (or objects that are instantiated once and accessed by multiple client objects), they can be members of many different patterns and architectures simultaneously. The mapping of such architectures includes the objects contained in the design as well as their interconnections, properties and other useful characteristics. Knowing which objects belong to which designs and hence systems is crucial to understanding how such systems are built and evolving. Static documentation will be challenged to keep up with dynamic systems like this.
The intent of this segment is to explore the use of sets and set notation as a primitive descriptive language and rule set for defining and understanding the scaled composition of complex arbitrary systems (arbitrary in the sense that a given collection of distributed objects may play roles in multiple systems and that they, themselves, do not represent one specific system).

Our previous discussions on *clusters*, *containment* and *multitude* lead us to believe that

> **Observation 5:** *"...set theory is a pure and reasonable method of describing the relationships between clusters, meta-clusters and their embedded patterns and designs."*

Understanding where these clusteral boundaries lie and how they are arranged maps cleanly onto the notion of sets. To this end, we wish to be able to understand the detailed composition of complex clusters and super-clusters and for this we need a language for discussing it. Sets and set notation are one such language. Our interest in using sets as a pure representation and language for describing logical constructs draws upon the following observation:

> **Observation 6:** "*Any conceivable pattern at all can be coded up by sets.*" [Rucker 95]

Given a landscape comprised of millions of objects and services, the benefit of being able to build many different kinds of systems and applications dynamically on top of those objects and services is undeniable. Being able to understand the overlapping topology of each system on this landscape is a difficult task, but not something that should be avoided due to difficulty. Our goal is to track the compositional topology as such systems are incrementally built, and to provide an underlying, pure form of representation (see section on *Complexity: Describing Complexity*) that will lead to useful visualizations of the boundaries between and within systems, separate from the various architectural or implementation-level visualizations, which may or may not exist.

The design process for massive fractal systems has yet to be invented, but suffice to say that a useful descriptive language will be required, much like UML is used today to build flat, static systems. Whether it will be based, in part, on set theory is unknown at this time, but for now, set theory demonstrates an ability to describe compositional structures like clusters which play an important role in our proposed architecture.

Fractal Pattern Composition

The geometric notion behind fractals is their self-similar *composition of structure.* Such structure is strictly derived from iterated calculations and hence purely mathematical in representation. Even though the world of OOP defies precise numerical analysis in that object abstractions and their implications derive from a set of human imposed requirements and not from some mathematically driven function, they *do offer* an explicit way of defining structure. This structure is evident in the interconnections, relations and compositions of such objects. For practical purposes in software development today, many such structures and composition semantics have been captured to address specific recurring dilemmas in OOP. Such reusable structures are appropriately called *patterns* [Gamma et. al. 1996]. The purpose of patterns is to apply well-known and tested solutions to common problems. The patterns defined in the text *Design Patterns: Elements of Reusable Software* are of a sufficiently low level that further decomposition is not likely. However, further composition at the macro level is not only likely, but obvious given the proliferation of discrete objectives in a given system.

As the objectives of a system are decomposed into finer and finer granules each sub problem can appropriately be addressed possibly in a generic or reusable way. This, indeed, is the goal of developing software with objects and patterns. Even if such a problem does not offer a generic or reusable approach, the systematic decomposition of functional requirements is an appropriate way to tackle large, difficult programming tasks.

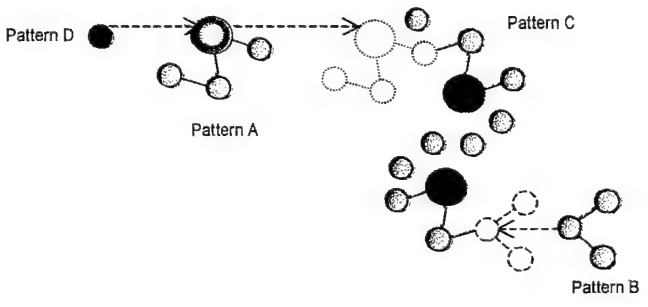
There are many sizes of recurring problems one may encounter, often depending on the system. Given this, it stands to follow that identifying the use of multi-granular patterns is valuable to the robustness and longevity of the proposed system as well as the reusability of solutions.

Viewing the composition of patterns into larger, higher-level structures mirrors the task of systematically decomposing the goals of a system and applying solutions to those goals until all goals have been satisfied [Gamma 1999]. Only through this systematic portioning of a systems objectives can one consider reusing or adapting any *portions* thereof.

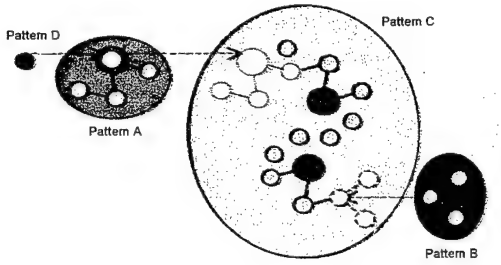
The quality of fractal inherent in this process is the presence of structure within structure. That is, large structural patterns composed of smaller structural patterns. The self-similarity is not strict in the true definition of the term, but similarity arises through the *process* by which such object patterns and compositions associate with higher level contexts. This process should be truly self-similar in that the composition and assembly procedures are irrespective to the scale of the pattern being composed. This consistency is beneficial to human designers as it reduces the amount of variation in procedures used to construct large systems by providing a uniform process for aggregating *structure.*

To illustrate this approach better, consider the following sequence of diagrams and their accompanying text.

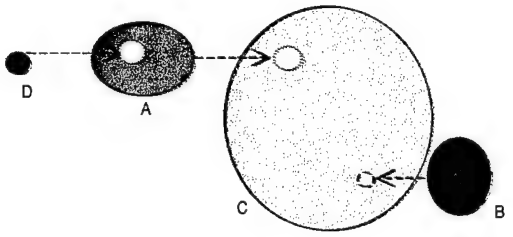
1. Patterns at three different levels or *scale* are represented in the graph below. We see that pattern **C** represents the largest complete pattern of objects of which pattern's **A** + **B** are components. Additionally, pattern **A** contains pattern **D**. From this organization, we observe that pattern **C** is the largest pattern and the aggregate of patterns, **B**,**A** & **D**.



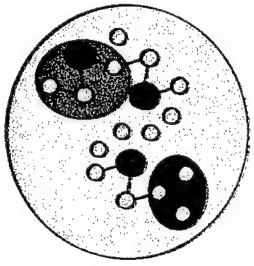
2. The diagram below has grouped each pattern into a discrete entity *with boundaries* rather than a multitude without boundary. In doing this, each pattern can now be manipulated more easily. The complexity within each pattern need not be unfolded simultaneously since we've allowed it to be handled as a *one* (see section on *The One and the Many)*. The diagram below, however, reveals each pattern's internal complexity.



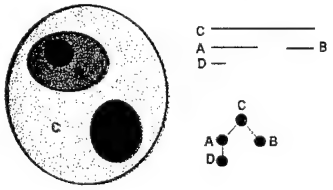
3. By hiding the details behind each pattern, the visualization is simplified for the observer. The manner in which these patterns interlock may still be well known or revealed through introspection, but from the designers perspective, details within these patterns are not important information at this magnitude and are not seen. Rather, the *interconnections* of these patterns into a further single entity is the interesting feature of this visualization.



4. Having created a new singular entity composed of multiple interlocked pattern components, we choose to reveal the depth of this design in the diagram below. Because it is a rather simple construct with a depth of only 3, it is quite easily understood; however, vastly more complex designs may exist and therefore, the viewing depth would need to be governed accordingly.



5. Finally, we set our viewing depth to reveal only the *organizational constructs* themselves and not their internal designs. This further limits the amount of information viewed and hence reduces the *perceived complexity* (see section on *Perceived Complexity*). Maneuvering between such levels of descriptiveness and complexity will be beneficial when constructing, understanding and viewing complex systems with similar structure (see section on *Complex Systems: Complexity Visualization*).



Representations & Codification Schemes

Given the architectural and structural concepts presented thus far, it is a major concern of this effort to discover a pure, objective representation for the relationships that, in unison, act as a given design, pattern or architecture. It is our hope and vision that the abstractions we visualize as designs can be thoroughly encoded into some *precise* and concise grammar, which is to say, a suitable alphabet that can accurately portray design relationships. It would seem, also, that the design relationships must be defined initially and given this, we expect only specific portions of our fractal design architecture to allow for this sort of descriptive representation at this time. The nature of relationships between entities in an architecture can have many semantic meanings and it is not our intention to define all such possibilities, rather we are concerned with the underlying composition, structure, and interconnections that encapsulate a design of a useful granularity.

Recently, we have been discussing the idea of containment graphs, clusters, sets and the notion of turning *multitude* into *oneness*. We presented the basics behind universal set theory in a prior section and have discussed the possibilities of fractal pattern composition. Let us now discuss the use of set notation as a means of representing or describing clusters and architectural compositions.

Using Set Notation

The decomposition of concerns (or problems) within a system and the subsequent design and application of discrete solutions or patterns that address such concerns bears a structure that can be expressed using set notation (as we posed earlier).

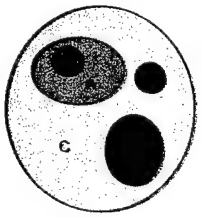
It can be said that for all concerns of a given design that the set C represents the set of all such concerns as a whole. Furthermore, the subcomponents of C themselves are sets which further subdivide into their respective subcomponents. The following expression captures this observation:

$C: \{ S(\chi) \}$

Where $S(\chi)$ provides the indication that χ is a component of (or composes into) C.

Therefore, for any given component of an architecture, be it a pattern, collection or individual component, it's organizational relationship to the whole can be defined and succinctly expressed by sets. It is indeed, the process itself of forming oneness from multitude that defines the very notion of sets and since this ability under rides our desire for scalability, size, and form it becomes a useful mechanism for representing and discussing such ideas.

Consider the following representation:



$C: \{A \in C: B \in C: D \in C \}$
$A: \{D \in A\}$

Without using open or free variables, we express the structure of cluster C by explicitly identifying its members.

A simple observation of this design can be:
$C \cap A = \{D\}$, which tells us that the intersection between A and C is the set containing D and from this we can conclude that both A and C are super-structures of D. Furthermore, the sum of D's containment can be represented by the sum of nodes where D appears in the containment hierarchy and therefore can be dynamically acquired through a simple recursive (self-referential) procedure.

The purest representation of sets can be expressed implicitly using only braces to indicate the containment boundaries between different sets. For example, the empty or null set is always seen as {}. Furthermore, the counting numbers are easily represented as follows:

1 {{}}
2 {{{}}}

3 {{{{}}}}

This method suggests that for any given number n, n^* (or the set representation for n) maintains n+1 left curly braces followed by n+1 right curly braces [Rucker 1995]. In other words, the *rank* of the set representation for a given number is equal to the number being represented. So for the number three, the *set rank* is also 3. It is indeed this simple isomorphism that allows all numbers and therefore all of mathematics to be represented as sets!

Given this approach, the sets themselves become rather anonymous. Only the purity of their relationships to one another is preserved. Using the previous diagram of patterns A, B,C, & D the following pure set representation accurately describes the organizational structure.

{ {}, {} , { {} } }

We could make the notation easier to translate by adding a label for each set before its first brace, such as:

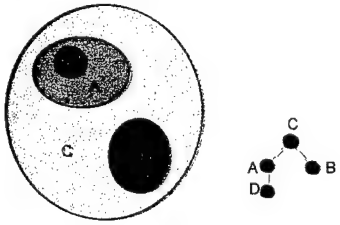
C{ D{}, B{}, A{ D{} } }

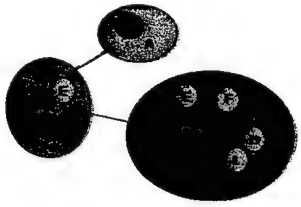
Here, we've simply prepended the corresponding pattern letter to the front of each set. It adds nothing to the organizational structure of the sets themselves, but only serves to help associate with the diagram. There are other such methods for describing and resolving this sort of containment structure which we'll look at.

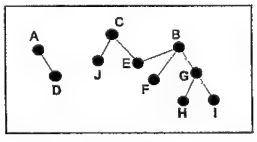
Containment Graphs

Since the most fundamental attribute between aggregate structures appears to be containment, we continue to adopt methods for capturing, describing and conveying that structure. These methods will ultimately lead to useful visualizations which is important for development of vast complex systems.

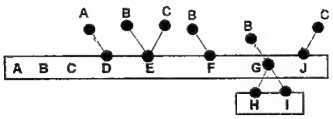
Similar to how we used pure set notation to convey the organizational structure of a given cluster, pattern or system, we can also indicate this using *containment graphs*. Like using sets, containment graphs are an easy concept to grasp. The relationship (or position) between nodes in a containment graph is one governed by composition. Specifically, nodes appearing above other nodes are necessarily the aggregates (or are composed) of the lower nodes. So, using the same diagram as before, we now add the simple containment graph which is analogous to the relationships conveyed by the original diagram. By simple inspection, it becomes obvious how the elements are interrelated in terms of structure. Namely, C contains A and B, while A contains D. Complex containment graphs can be pruned to provide isolated views into a complex system. Furthermore, the depth of nodes within a graph can also be incrementally expanded to reveal levels of complexity.







Entity containment graph

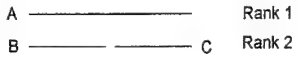


Entity enumeration map

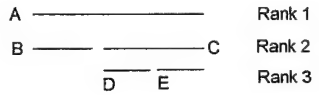
Segmented Line Glyphs

Another interesting way of presenting scaled compositions are *segmented line glyphs*. Equal in expressiveness to both sets and containment graphs, line glyphs are just another manner of conveying the same information. Line glyphs attempt to capture the containment structure into a more succinct symbolic format. Where set notation relies upon language based symbols as part of its grammar, line glyphs (and containment graphs) provide a non-language way of conveying structure. Unlike containment graphs, line glyphs are not joined or connected entities and because of this they are not easily navigated internally. Rather, the composition of structure stems from the upper most layer towards to lower most layer. Each line representing a set at a particular rank. Sets embedded within sets appear below their supersets. This is a recursive (and fractal) visual structure quite similar to Cantor's Set [reference]. Line glyphs offer very high compression ratios meaning that because they can be infinitesimally shrunk they can occupy very small spaces (including those not visible to humans). The lengths of line segments in a SLG need not adhere to a specific precision, but the sum of members at the same rank should never exceed the length of the parent rank. To do so, would disrupt the visual continuity.

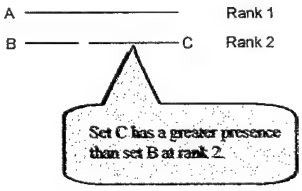
To clarify, consider the following simple set A with two elements, set B and set C. The line glyph representing this structure is as follows:



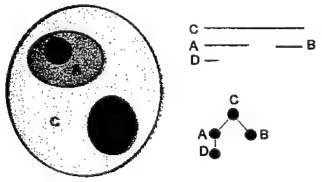
It should be easy to see that the top line is composed of two elements with equal fervor. If we were to add members to C, say set D and set E, we would end up with.



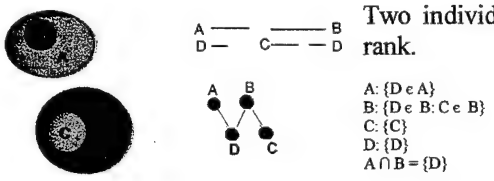
Now that set C has two compositions, namely D and E, it's interest is expanded below A and hence it's line presence grows to accommodate. This provides useful information to the observer at rank 2 that indeed, set C has a larger presence.



The following diagrams show three identical forms of set containment representation.



A single set with depth and rank of 3.



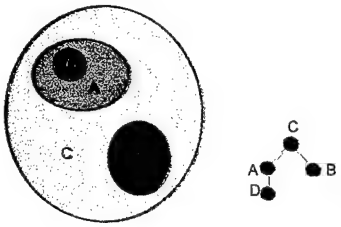
Two individual sets with a common component and rank.

XML (Extensible Markup Language)

XML (eXtensible Markup Language) is a dynamic hierarchical descriptive language used to describe any type of entity or abstraction. XML would be quite useful in describing a recursive or embedded patterned structure as it allows for vast complex trees of objects to be described and linked to one another in interesting ways.

By using special tags that apply to architectural and fractal constructs, an appropriate XML string can be used to represent those constructs. This would include all state properties of each entity being represented. Through state decomposition, a structured *descriptive string* can be acquired. This description can be used to store the system state, visualize the system with new visualization tools or re-instate a given system from its previous description.

Also, XML is attractive as a means of interoperability between network services, applications and businesses. If multiple participants are involved in constructing or contributing to large projects and systems, possibly using their own tool sets, then encoding certain aspects in a common, open format will allow them to utilize the representation in uniquely important ways without requiring special demands from the entire community.




```
<CLUSTER NAME=C ORIGIN=&address>
  <PROPERTY KEY=VALUE>
  <PROPERTY KEY=VALUE>
  ....
  <MEMBER>
    <CLUSTER NAME=B ORIGIN=&address>
      <PROPERTY KEY=VALUE>
    </CLUSTER>
    <CLUSTER NAME=A ORIGIN=&address>
      <PROPERTY KEY=VALUE>
      <MEMBER>
          <CLUSTER NAME=D ORIGIN=&address>
          </CLUSTER>
      </MEMBER>
    </CLUSTER>
  </MEMBER>
</CLUSTER>
```

Designing in the Large
Designing in the Small

The concerns of a complex software system are manyfold and not entirely known in advance of production. The more complex the system, the more concerns it is likely to have. More often than not, these concerns will span a spectrum of scale in that some will have less of an impact than others. In this regard, we can gauge the effect of a particular concern, some being large and some being small. Large concerns tend to be different in nature than small concerns and often require different designs and technologies . Often there are different individuals responsible for large and small concerns. For example, deciding the best algorithm to use for sorting a double-linked list may be viewed as a lesser or smaller concern than deciding on how two systems might exchange transactions. It would seem that even concerns of a system fall into an embedded composition structure, with larger more important concerns lying near the top. Such large concerns may have many embedded concerns as well.

Whether you equate the concerns of a system with requirements, tasks, problems or whatever else, it should be clear that they offer themselves in such a way that a spectrum of scale emerges. From this, we consider a design approach to a complex system that recognizes this spectrum. The ideas of designing in the large vs. designing in the small have appeared in software engineering lately. These phrases are meant to separate the vastly different objectives that lie at opposite ends of the scale spectrum of concerns. As system engineers, we cannot ignore the entire spectrum, but discover that teams of people are needed to cover its entirety.

Part of this stems from the lack of uniformity surrounding software development. The multitude of technologies involved in a complex system are often so disparate as to impose the spectrum themselves! It should follow from these observations that new technologies or tools that can allow fewer people to more easily control the concerns across the spectrum and in a more dynamic and spontaneous way would be a huge step forward in software system construction.

We foresee the ability to readily perceive a spectrum of design and organization for a given system. This ability will be brought about by tools and technologies we intend to innovate that will permit more seamless interaction between large and small concerns. This is to say, that one should be able to design *in the large* similar as *in the small*.

Design Elements

This section introduces some of the interesting design elements and constructs resulting from our current investigations. These objects will likely transition into our Core Technology Framework which will provide the necessary foundation for constructing and adapting scalable fractal-like architectures.

The design elements discussed purposely offer no particular implementation and as of yet, only generalities surrounding their possible implementation may be known. Here, our goal is to define the higher level abstractions foreseen to be useful in achieving the prime objective separate from any given technology driving it. This will allow us to define a pure set of abstractions from which we can build or evolve a specific implementation using the most advanced technologies available and inventing others.

Object

Just as with any object-oriented programming environment, the object is the fundamental building block. Since we intend to utilize OOP practices and methodologies on this effort, the object is the natural basic unit. The concept behind the object is rather simple, that is, a discrete entity with both properties and behaviors.

Aside from the ability to program objects in a programming language, our system will have the ability to create objects dynamically. The need for this stems from many areas. Some of the foreseen dynamic object types include:

- Dynamic adapters to connect components
- Dynamic message handling objects
- Dynamic interface implementations
- Dynamic cluster or pattern objects

Regardless of the scale factor within our architecture, any given entity can be succinctly represented as an interface or pure object. The ability to interact with a system regardless of size or complexity will manifest through simple object interactions. Whether a given object represents an entire system or a tiny component should not cause a variation in procedure.

It is also worthy to mention that the term component is used quite frequently and with regards to these design elements, components are indeed objects.

Node

The *node* element represents the fundamental or logical "place" where an object resides. Running objects reside within nodes. Each node is bound to a physical location on a network, yet many nodes may be running on the same physical computer. Nodes provide some very important runtime services to application objects contained within them. Some of these duties include:

- Dynamic object registration
- Registry Services
- Proxy/Object persistence
- Network Awareness
- Dynamic Proxying

Nodes provide the necessary topological infrastructure through which components, patterns, applications and systems can be created deployed and ultimately linked together and interact. Each node will provide its own object request services allowing objects across nodes to communicate dynamically. Furthermore, the footprint for nodes will support small resource-deprived devices as well as large super-computers.

Registry

The *registry* is one of the most important objects within this architecture. It provides the ability for application objects to discover and connect to other application objects, services, or components dynamically and without *a priori* knowledge about the availability or physical location of such objects.

The registry behaves like any registry you would expect in that information in the form of objects and directory structure is created and posted as entires for subsequent lookups. However, this registry contains some very unique capabilities (see section on *Federated Object Registry*) some of which include:

- De-centralized federation
- True fault-tolerance
- Dynamic visibility and awareness
- Registry scalability and linking
- Hierarchical composition

The registry is the lifeline of an adaptive system as it allows for components to come online and post their membership to, or departure from, a system dynamically. The notification mechanism for this is based on a scalable dynamic multicasting protocol that informs all interested components and applications so proper reaction and adjustments can occur. This is an important feature in a complex distributed system that can experience frequent or multiple partial failures and must continue operating through such periods.

Since the registry is based on a federation of independent registries it strictly does not employ a replication strategy. There are a few very important reasons for this.

1) It is not assumed that the size of a given node can accommodate a large replicated registry.
2) Since all objects are serviced through their nodes, and each node hosts its own registry, duplicating a given registry in other nodes does not guarantee that the original object is accessible if its node goes offline. In fact, with our approach, if the node is offline, so is its registry and therefore no successful lookups can occur for those objects contained in that node.
3) With the ability to link systems together, it would not make sense to promote replication of registries across boundaries where those registries may not be useful. Creating an automated process for controlling this would be difficult or impossible and additional human intervention would be needed in these cases.

Cluster

The *cluster* represents the fundamental organizing construct through which scaled compositions and structures can be created. A cluster is an artificial (or logical) boundary grouping together a multitude of objects into a single body (see section on *The One and the Many*). As a mechanism to control complexity, clusters can be used by designers to organize and structure a vast multitude of objects to harmonize with various intrinsic or extrinsic properties. Clusters allow many objects to be viewed and manipulated as a whole or a one.

Objects become members of clusters by announcing their desire for inclusion in a particular cluster. Since the cluster itself exists in no single location, it is the sum of all members that becomes the unified cluster. However, special property facets can be attached to clusters that govern and control how and when objects are allowed to become members. Furthermore, the visibility of a cluster can be governed by these property facets. Once created, a property facet will listen and respond to announcements received regarding the membership or departure from a given cluster. Advanced tools we intend to develop will make creating, interacting with and viewing cluster objects intuitive for humans.

Clusters can themselves contain other cluster references. This allows for the organized discovery and traversal of clusters and their members. Dynamic multicast messages can be sent to a cluster as a single object and the message will be received by all of its active members transparently.

Domain

A domain is a special type of cluster object that has specific properties useful in constructing large bounded domains of interest and information. A domain represents a logical boundary containing users, applications and data of a common goal or interest. For this reason, domain objects have additional properties regarding their visibility and security requirements controlling access and interaction to elements of the domain.

A domain can be partitioned into many different levels of interest allowing for dynamic organizations of users and data. This is necessary where there exists an intermingling of users who utilize both public and privileged data and services. Although one can construct segmented compartmentalized domains, often, a domain will need to cater to many different security levels of interest simultaneously. The exact structure and partitioning of a domain into these areas is entirely up to the designers.

Event

An event object is responsible for transmitting information from one object to one or more objects. Only through events does information propagate throughout a distributed system. The change brought about by events is represented in the data contained within the event. Application designers have full discretion as to what constitutes an event in their system. However, the infrastructure will have many of its own events that occur to notify itself of happenings within the system and handle them automatically.

There are different ways an event can be delivered to accommodate a specific need of a system. Some of these include:

- Direct synchronous
- Direct asynchronous
- Store-and-forward
- Unreliable Multicast
- (Reliable Multicast)

Architects will have the ability to specify the nature of particular events and event connections between components dynamically at runtime. This will allow for appropriate dynamic optimization of communication channels in a system without redesign efforts.

It is expected, also, that the messaging facilities provided by the framework do not articulate any specific message or event implementation. For this reason, appropriate interfaces to other messaging/event services can be accomplished. By providing a transparent and configurable bridge to interfaces such as JMS (Java Messaging Services), events generated in a system can be sent to other services over a particular messaging implementation such as MQ series or others. Furthermore, objects within the system should be equally unaware about the exact nature of the message implementation when sending or receiving events across a channel.

Channel

A channel is similar to any other type of channel whether TV or radio. Within the infrastructure, listeners can listen to a specific channel and hear multicast event messages that are transmitted over the channel. The listener must pass any authentication requirements on the channel in order to be a member. Providers will post event objects onto channels which are then instantly received by all current listeners. This form of delivery is called the publish/subscribe model.

Publisher

A *publisher* is a type of object that posts events onto a channel. The publisher need not be aware of the number or existence of clients listening on the channel. The Publisher interface may be implemented by application objects to provide their own specific publisher functionality.

Subscriber

Subscriber objects listen or "subscribe" to event channels and in doing so receive events as they are published to the channel. All subscribers to a channel will receive the same multicast event objects as they are transmitted. Application objects may choose to implement the Subscriber interface. Default implementations will be provided by the

framework and utilized within builder tools to provide the necessary functionality transparently to application beans.

Container

Container objects wrap or contain application objects and provide appropriate infrastructure level services to those objects transparently. Containers are responsible for storing lists of facets (e.g. service or property) associated with a given object. Containers can also operate dynamically on application objects such as generating a runtime proxy, cloning or persisting the application object on demand.

When an application object is relocated across a network, its container is also relocated. The container provides negative-space functionality to the object within (see section on *Negative Space*). The container and its object are essentially bound together. The architecture infrastructure will interact with an objects container affecting certain meta-level properties and performing useful operations without any coding effort on behalf of the application object. Furthermore, additional functional objects can be placed inside of containers dynamically to extend the capabilities of the infrastructure and hence the application system dynamically.

Metadapt visual development tools will allow designers to connect their application objects to an existing distributed system via its container with little or no re-coding required.

Gateway

The Gateway object is a special kind of object associated with channels that permit them to be bridge across clusters, domains or systems. Gateways contain the policy requirements needed by other systems to publish events on the given channel. Such policies might contain authentication restrictions or access control lists based on IP address, subnet mask, digital certificate, user/password or other common verification methods. Gateway's will utilize a variety of facet objects to achieve certain functions such as security, reliable asynchronous messaging and others.

Objects wanting to attach to or interact with other system objects will typically be authenticated by a Gateway if one is present. This process happens automatically and need not be hard-coded into specific application objects.

Facet

Facets are special negative-space objects that attach dynamically to application components when they are registered and deployed into a system. The architect has the ability to specify which added functionality he/she wishes to be attached to a particular object when it is deployed. Some service facets will be automatically attached. These facets will handle the necessary functionality of discovery, notification, security and proxying to name a few.

Service

The service facet is responsible for interfacing the component to the infrastructure services. When a component is deployed, a service facet is attached and properties about the component are assigned or derived and stored in the service facet which transmits the appropriate information to the architecture community using special multicast discovery and announcement protocols.

Event

The event facet is a special adapter that links event notification between two distinct components. Each event facet has a cardinality equal to the direction of the event flow between the two components. Therefore, for any two components, each will have an event facet representing the outbound and inbound properties of the connection. Tools inspecting the negative-space will be able to deconstruct relationships between components by traversing their event connections. The objects themselves are unaware of these facets and do not need to explicitly provide support for them.

Publisher The publisher facet can be attached to an object to provide publish/subscribe capability. When the application object is linked to a publisher facet it can send events to multiple listeners simultaneously. Each publisher facet can be configured separately. The publisher facet implements the Publisher interface.

Subscriber The subscriber facet is attached to an object wishing to receive events off a channel, but does not provide an implementation to Subscriber directly. Rather, it receives events from the Subscriber facet which is configured to authenticate and listen to a particular channel.

Security The security facet is responsible for holding and passing digital certificates, encryption algorithms and other similar security objects and properties. Application objects requiring security features can have these facets configured and attached to them dynamically. For this reason, any object can be assimilated into a secure environment, given an authenticated security facet.

Property This facet is responsible for containing and returning meta-data about the component it is attached to. Tools operating on the infrastructure will inspect and visualize components based on the meta-data properties supplied by the property facet. Property facets can be coded and provide by developers to suite their system needs. Generic property facets are provided that contain simple key-value lists of properties that can be configured at runtime by tools operating on the infrastructure negative-space.

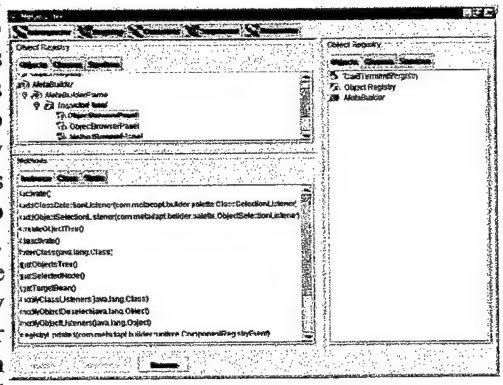
Software Tools

It is our ultimate vision that sophisticated new tools will be created that layer upon our lightweight framework. During this Phase I we have been prototyping one such tool called *MetaBuilder* (MB) which demonstrates some of the concepts and ideas we've been developing. We hope that many more useful tools can be developed to help make this new architecture paradigm powerful and easy to use.

MetaBuilder

The term *meta* means *above* or *outside of.* Our tool, MetaBuilder, is a type of advanced component and system assembly program for building, deploying and adapting applications. It allows for the dynamic instantiation of objects and components. Furthermore, it permits runtime interaction with such objects exposing both their properties and behavior. Through this, objects and components can be connected together to create usable systems and applications very quickly.

Because the objects loaded into the MetaBuilder registry do not need to extend a certain class or implement specific MetaBuilder interfaces, there is no intrusive requirements for using objects and classes within MetaBuilder. In this sense, the objects do not know they are being manipulated as such and therefore MetaBuilder truly is *above* them. There are many advantages to this, the most obvious being that any qualified class can be used in this environment and subsequently linked into applications and systems dynamically. Furthermore, MB will allow for the dynamic creation of complex arbitrary method calls across components (either local or remote). These *call sequences* can be linked to events triggered by individual components (via the JavaBean event model). Any number of method parameters can be provide either statically or through nested invocations to other components local or remote and encapsulated as a first-class object itself for future manipulation, reuse, or wiring.



Objects, beans or components loaded into MetaBuilder are dynamically processed to permit for network transparency and mobility (insomuch as interacting with properties and behaviors remains constant regardless of network location). What this means is that any plain Java object can be loaded, instantiated and remotely deployed (to another machine) while it is running. Any references to that object will continue to operate as if it were a local object even though it's remote location may continually change. The connection properties of messages sent to this object can be manipulated independently of the object themselves freeing designers from deciding (or knowing) a priori how to handle network issues. For example, one can change the connection between two components from direct/ synchronous to indirect/asynchronous/reliable. Because the connection properties are managed in the negative space, the objects themselves do not need to mandate a concrete policy.

This allows applications distributed or otherwise to be created on a single desktop and then deployed over a network where they now run in a distributed fashion. Components of the system may still have their properties altered or methods invoked while they are running. Future objects can easily be connected to existing deployed objects thereby extending the functionality of systems and applications quickly and easily. Furthermore, objects already running on the network can be re-inspected, modified, removed or otherwise changed on-the-fly.

MetaBuilder will support a variety of object types including but not limited to:

- Any JavaBean or Java Class
- Enterprise JavaBeans
- RMI Objects

- CORBA Objects
- COM Objects
- Voyager Agents

Or any JNDI provided object where the designer can plug his/her own JNDI service provider directly into MB.

Integration with the One Registry

MetaBuilder will provide access to not only its locally running registry, but will also provide tabbed panes for other lists of objects like the Java 2 platform classes and the **One Registry** (see section on *Federated Object Registry*). Through the One Registry, MetaBuilder can see all registered objects running across a network or between networks giving the designer the ability to connect to services and remote objects dynamically. This ability integrates the efforts of other individuals, teams or sites. As objects and services become available or become unavailable, MetaBuilder instantly recognizes this and the designer can adjust accordingly., knowing which components and services are available at any given time.

Not only does the application designer have his/her own component libraries to work with, but the published components and services across networks as well (see section on *ObjectCore*).

Integration with ObjectCores

An ObjectCore is a special site (build from our architecture) that acts as a repository for component classes, applications and actual running objects and services. Each ObjectCore publishes the objects and classes for use by designers over a network much like internet sites today publish static HTML documents for viewers of the internet. Through integration with ObjectCore sites, MetaBuilder will be able to tap into specialized developmental repositories all over the world. Directly from MB, the designer can view many different ObjectCores and the services each has to offer and spontaneously build applications or connect existing applications to those services along with locally available resources. Furthermore, component classes might be published as well, allowing the designer to dynamically download new classes to instantiate and connect immediately into running systems and applications.

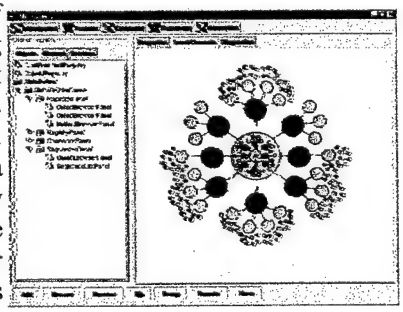
Integration into Applications

MetaBuilder is designed to not only function as a stand alone tool, but also to be embedded into Java applications similar to how Visual Basic Editor for Applications operates. However, with VB, only the current application can expose its own objects for wiring. Using MB, the designer can see not only the current application from which it was launched, but can also see the local network and ObjectCore's around the world! This allows a user to connect particular applications together dynamically without those applications having to support any particular API or interface other than launching a MetaBuilder and/or registering their objects in local ObjectCores or One Registries.

For this to happen, the designer chooses which internal objects of a given application are to be exposed and registers them in the local MetaBuilder static registry. Once MetaBuilder is launched from within the application, registered objects and services can be seen. From there, the user can connect the application objects to other beans or services (like Jini) dynamically.

MetaViewer

MetaViewer is a tool similar in concept to MetaBuilder in that in remains *above* or *outside* of the system itself. The goal of MetaViewer is to leverage *form* existing in the negative space (provided by our framework implementation) to accurately visualize systems built with and deployed from MetaBuilder. Therefore, the same pure objects deployed into a system from MetaBuilder can now be easily visualized and interacted with by MetaViewer. We have not done any development or prototyping for this tool and would hope future contract efforts would provided the necessary funding to do so.



Commercial Concepts & Strategy

Our commercialization strategy for the resulting technology developed with our current and future SBIR funding covers a wide range of use and vision. Our ultimate goal will be to leverage the size and growth capacity of the internet and corporate intranets by providing a valuable environment for users to work within as well as a scalable fractal architecture with which to build adaptive systems driving it all. This end-user experience approach to providing the motivation for compelling systems to be created with our technology is the same approach seen during the early days of the internet. Namely, web browsers were easy to get and free, subsequently web sites and server software became an obvious attraction to commercial enterprise.

The same basic model of interaction with the internet occurs now as it did before. That is, a web browser dynamically attaches to a site and acquires HTML documents, which it renders. Sites can come and go dynamically and users need only point their browsers at a site to scan it's content. The rendering produced by HTML is more content driven with only a handful of fundamental objects designed for acquiring information and handling user interaction or input. Despite this drawback in comparison to "desktop" applications, the internet, its services, applications and advertising is ever more popular and useful.

However, the container we all use to view and interact with the internet, namely the browser, is essentially an HTML renderer and HTML is a markup language, not a programming language. Web apps lack many of the capabilities and user-friendly features of "normal" applications we are used to using. Nonetheless, because we can surf and acquire web-based HTML apps dynamically, they are immediately valuable, if not all that usable. Furthermore, such web-based apps are typically freely available through internet portal sites driven exclusively by advertising income.

These are concerns we see advancing in the future. Namely, not only will we surf web-sites for content (using our current browsers), but we will also be able to browse and acquire micro-components, tools, applications and services as well. The latter focuses more on *functionality* and *action*, while the former deals with *content* and *processing*. A rich, equally dynamic environment where both content and functionality can be dynamically discovered, acquired and used to solve tasks is where we see the internet evolving to. In fact, it is already getting there.

No longer are ISP's solely intent on providing internet *connectivity*, but are now providing internet *productivity* as well. With such web-based HTML apps as schedulers, calendars, mail, and others, ISP's and portal sites are offering tools that can be used anywhere on the web. However, such tools are currently coded in HTML on the front-end and therefore quite limited in their usability and appearance. We see this transitioning to a richer end-user experience that can provide huge leaps in capability as well; one provided by a platform neutral, internet ready language like Java. Therefore, a stable, secure and reliable mechanism must exist to provide the dynamic discovery and acquisition needed to overstep current HTML based web-tools (but not necessarily obviate them). Furthermore, the current HTML-based productivity apps are distributed free-of charge, but are often slim on features and are always accompanied by a barrage of advertisements. We see the rise of ecommerce reducing the reliance on advertising money to provide users with tools and services they're willing to pay for at extremely small, volume driven costs or bundled with ISP services.

Companies like BizTone are already capitalizing on Java and it's related technologies to provide dynamic *managed application services* over the internet to business customers who wish to completely outsource the development, control and management of business critical applications. Rather, these businesses adopt a more economical pay-as-you-go paradigm for using these new application services. This way, small businesses can choose services economically important to them and graduate to larger systems easily without worries about IT infrastructure. The overall cost of transition, IT and miscellaneous overhead is virtually eliminated with this new model.

Although we see great potential in this approach to businesses, we see a much larger potential in sheer volume with internet users in general. In the future, users will be more

savvy and utilize a variety of specialized tools on their desktop to perform internet duties than just a simple browser. Such applications and tools, however, will be acquired just as easily as today's HTML based apps are accessed via the web. Tools can be easily docked on a users desktop where they have easy and constant access to them. Sites providing such tools can manage versions for customers dynamically and transparently. Furthermore, such sites specializing in providing usable tools, components and applications for internet productivity can do so with the benefit of ecommerce. This will allow users to acquire and use application services on-demand and pay only micro-values for incremental use of such tools (e.g. 1/50 of a penny per transaction).

For example, one might acquire the new Yahoo! search agent application, dock it, and spend a day using it to gather information. The entire day's use might cost $1.50 at which point the user may never need to use it again. Multiply this scenario by millions over time. Many users may only need such services briefly and would not spend $50 to buy packaged software, whether at a local store or online where, in either case, a day or two delay might be involved. Also, the freely available versions may not be feature-rich and certainly would not compete with off-the-shelf versions anyway. We refer to such low-cost volume centric services as *micro-value services*. Furthermore, sites specializing in micro-value services will have their services bundled and aggregated into yet other micro-value services as new markets for specialized data and applications based on such services continues to rise. This scaling effect is already present within standard economics. Once global automated ecommerce becomes widespread, such micro-value services will not only proliferate, but users will be able to tap into world's knowledge base of information in more dynamic and productive ways than is currently possible.

Our architecture and tools will provide the *landscape* through which such a vision can be realized. The fractal qualities of networks, economics, communities, bureaucracies, and other areas require the underlying technology to provide and equally fractal and scalable isomorphism. Therefore, such inherent qualities should not be forsaken by any technology attempting to accurately reproduce massively scalable form and function. We feel that motivations behind our lightweight node-based architecture with it's fractal qualities of structure and composability will provide the foundations for such endeavors, as we've described, to accelerate. Furthermore, we intend to develop a valuable, pervasive end-user experience called the **Internet Desktop** that will act as the new wave internet portal for power users of tomorrow. This new type of portal will be driven not by unconnected static content (as it is now), but by dynamic interconnected components, applications and ecommerce services enabling users to raise the bar on productivity from the internet.

ObjectCore

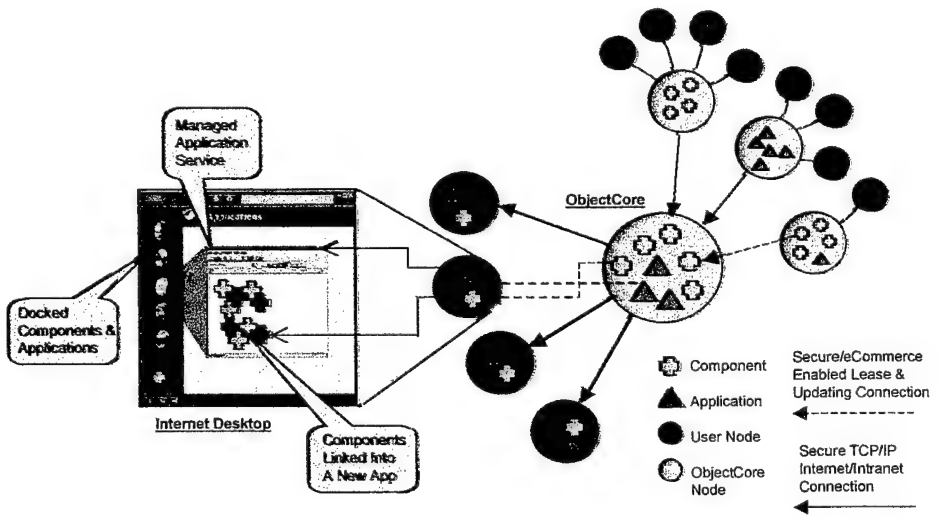
An ObjectCore is a internet repository for managed application services, application components and other similar types of objects. ObjectCores will be visible across the internet or within corporate intranets and serve to warehouse and deliver state-of-the-art application tools and components to end-users who will dock them on their "Internet Desktop". The idea is that users can connect to many ObjectCore's and decide which tools, services and components they want to acquire or otherwise tie-into and use. Since eCommerce is built into each ObjectCore, users can begin using its services immediately and only pay micro-values for use of such services much like leasing or renting except on a very small scale. ObjectCore's can aggregate objects and services from other ObjectCore's and provide layered value-add to their specific user or customer bases as well.

Component & Application Repositories

The internet of the future will provide more than just static content and searching capability; rather, valuable commerce related services (currently residing on the back end of web systems) will also be available for users and business to "tap" into and use in their own applications and tools. Furthermore, sites providing more than content will find it useful to create specialized tools and applications for their users who will need a way to acquire and use them dynamically. This entire process will be automated. Sites will provide quality tools and services which users will acquire and *dock* on their Internet Desktops and hence build a specialized tool set optimized to the way they work and the kinds of tasks they perform. Furthermore, the use of such tools will be governed by micro-value transactions that make it economical for users and profitable for companies.

Managed Application Services

Each ObjectCore will publish its set of applications and tools for users to use and consume. Since this process is entirely automated, an ObjectCore must ensure that users have software updates automatically, that is, when a newer application is available, current users of the application have the copies on their computers updated automatically. This technology has enabled a new internet market to emerge called *managed application services*. Companies not wishing to bear the cost and overhead of buying, building and managing their IT software systems can acquire them from companies who provide them across the internet as services. Companies like BizTone are already doing this with business financial applications. In the future, many more managed application services will emerge; not only for business, but for internet users as well. A common paradigm, architecture or system to do this would certainly be beneficial and is where we see the greatest commercial potential for our architecture and toolset technologies.



World Objects

Once ObjectCore sites begin to proliferate, end-users and developers will have many options about applications, objects and components they can acquire and use or build with. Considering this on a large scale, we will see that at one's fingertips emerges a library of useable and reusable components that can easily snap together into specialized tools and applications. The moniker we've chosen to describe this *common* repository and acquisition method of distributing objects and using them to build useful tools is *World Objects.*

A Global Component Toolbox

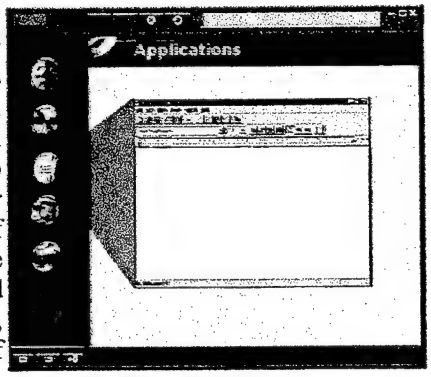
In the future, one will not be limited simply by the component libraries bundled with the current version of his/her IDE, but will find an expanse of useful, managed services and components throughout the digital world from which to choose from. Since these components are governed by micro-value transactions, they are cheap to buy/use and profitable to develop.

Developers will be freed from the component toolkits loaded on their computers and will truly have a global component toolbox from which to build from. As sites continue to post useful agents, components and services to assist in the use of their data, developers of the future will have an ever growing API or toolkit through which interesting, useful and dynamic tools and transient apps can be built. This will promote the idea of end-user programming as savvy internet users of tomorrow will create a rich working environment themselves.

Internet Desktop

The end-user experience to all of this is provided by the **Internet Desktop** (ID). Much like your regular desktop contains the applications loaded on your computer, your Internet Desktop will contain docked applications, components and services. But unlike your traditional desktop, your Internet Desktop goes where you go and can be accessed across the web.

As a user browses ObjectCore sites and selects objects for docking, the objects are downloaded and installed on the users local machine; however, they are maintained and managed by the ObjectCore. Execution of said applications occurs on the users machine, but new versions are managed by the ObjectCore and updated automatically. Furthermore, accessing one's ID from another computer will still display the users desktop, but accessing applications will cause them to be loaded and installed on the new computer dynamically. This allows one to access their ID anywhere and get to their applications. As the bandwidth and speed of computers and the internet rise geometrically, the transfer and/or access times of applications and components will diminish; cable modems, broadband wireless and DSL are examples of technologies that will make this happen.



Note: The images used to represent the ID were borrowed from various places and is not intended to represent an original work of art, but merely a quick conceptualization. Any infringements on design or image is purely unintentional.

Component Application Building

In addition to browsing application service sites with ID, users will be able to mix and match components from these applications into new applications on-the-fly [see section on *Adaptable Software*]. Lightweight MetaBuilder technology will be embedded into ID and allow for docked components to be interconnected and integrated into existing applications to provide richer and more accurate functionality. Furthermore, applications can be augmented with new components over time as desired by the particular user needs.

References

[Alexander 99] M. J. Alexander, Contributing Author, R. Hoque, Author, *XML for Real Programmers*, J. Wiley & Sons, Inc., New York (Available Fall 1999).

[Bar-Yam 97] Bar-Yam, Yaneer . "*Dynamics of Complex Systems*". Reading, MA. Addison-Wesley, 1997.

[Ben-Shaul 98] I. Ben-Shaul, J. W. Gish and W. Robinson, "An Integrated Network Component Architecture", *IEEE Software*, September/October 1998, pp. 79-87.

[Berg 99] D. J. Berg and J. S. Fritzinger, *Advanced Techniques for Java Developers*, J. Wiley and Sons, New York, 1999.

[Booch 94] G. Booch, *Object Oriented Analysis and Design With Applications, Second Edition*, Addison-Wesley, Reading, MA, 1994.

[Bradshaw 99] J. M. Bradshaw, M. Greaves, H. Holmback, T. Karygiannis, W. Jansen, B. G. Silverman, N. Suri and A. Wong, "Agents for the Masses?", *IEEE Intelligent Systems*, March/April 1999, pp. 53-63.

[Brooks 87] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering", *IEEE Computer*, April 1987, pp. 10-19.

[Carriero 90] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. The MIT Press, 1990.

[Clark 99] D. Clark, Service with a (Smart) Smile: Networks Jini Style", *IEEE Intelligent Systems*, May/June 1999, pp. 81-83

[Cline 96] M. P. Cline, "The Pros and Cons of Adopting and Applying Design Patterns in the Real World", *Communications of the ACM*, October 1996, pp. 47-49.

[Cockburn 96] A. Cockburn, "The Interaction of Social Issues and Software Architecture", *Communications of the ACM*, October 1996, pp. 40-46.

[Cohen 94] J. Cohen and I. Stewart, *The Collapse of Chaos: Discovering Simplicity in a Complex World*, Penguin Books, New York, 1994.

[Cooper 98] J. W. Cooper, "Using Design Patterns", *Communications of the ACM*, June 1998, pp. 65-68.

[Dreyfus 98] P. Dreyfus, "The Second Wave: Netscape on Usability in the Services-Based Internet", *IEEE Internet Computing*, March/April 1998, pp. 36-40.

[D'Souza 99] D. F. D'Souza and A. C. Wills, *Objects, Components and Frameworks With UML: The Catalysis Approach*, Addison-Wesley, Reading, MA, 1999.

[EDCS 99] Evolutionary Design of Complex Software (EDCS) Program, sponsored by DARPA, http://www.sei.cmu.edu/community/edcs/

[Fayad 96] M. Fayad and M. P. Cline, "Aspects of Software Adaptability",

Communications of the ACM, October 1996, pp. 58-59.

[Gamma 95] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addions-Wesley, Reading, MA, 1995.

[Gamma 99] Gamma, Erich , "*JUnit:: A Cook's Tour*" , JAVA Report. May 1999. pp 27

[Gazale 99] M. J. Gazalé, *GNOMON: From Pharaohs to Fractals*, Princeton University Press, Princeton, New Jersey, 1999.

[Gleick 87] J. Gleick, *Chaos: Making a New Science*, Viking-Penguin, New York, 1987.

[Glinert 90a] E. P. Glinert, Editor, *Visual Programming Environments: Paradigms and Systems*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Glinert 90b] E. P. Glinert, Editor, *Visual Programming Environments: Applications and Issues*, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[Govoni 98] Govoni, Darren , "*Composite Foundation Architecture: A fractal approach to scalable OO systems*", 1998.

[Govoni 99] D. Govoni, *Java Application Frameworks*, J. Wiley & Sons, Inc., New York, 1999.

[Greenberg 99] I. Greenberg, "Facing Up to New Interfaces", *IEEE Computer*, April 1999, pp. 14-16.

[Gupta 98] A. Gupta, C. Ferris, Y. Wilson and K. Venkatasubramanian, "Implementing Java Computing: Sun on Architecture and Application Deployment", *IEEE Internet Computing*, March/April 1998, pp. 60-64.

[Halmos 60] P. R. Halmos, Naïve Set Theory, Springer-Verlag, New York, 1974.

[Harel 92] D. Harel, "Biting the Silver Bullet: Toward a Brighter Future for Software Development", *IEEE Computer*, January 1992, pp. 8-20.

[Hendler 99] J. Hendler, "Making Sense out of Agents", *IEEE Intelligent Agents*, March/April 1999, pp. 32-37.

[Hofstadter 79] D. R. Hofstadter, *Godel, Escher, Bach: an Eternal Golden Braid*, Vintage Books, New York, 1979.

[Hölldobler 1994] Hölldobler, Bert ; Edward O. Wislon; "*Journey to the Ants: A Story of Scientific Exploration*". Cambridge, MA. Harvard University Press, 1994.

[Iowa 99] Complex Adatptive Systems Group at Iowa State University, http:// www.cs.iastate.edu/~honavar/alife.isu.html

[Janssen 69] W. C. Janssen, "A Next Generation Architecture for HTTP", *IEEE Internet Computing*, January/February 1999, pp. 69-73.

[Java 99] Java 2 Platform from Sun Microsystems, htp://java.sun.com

[JavaSpaces 99] JavaSpaces Technology Kit from Sun Microsystems, http://www.sun.com/consumer-embedded/cover/jstk-990615.html

[Lear 99] A. Lear, "Battle Begins for Device Network Supremacy", *IEEE Computer*, June 1999, pp. 20-21.

[Linda 99] The *Linda Coordination Language* homepage for the Linda Group at Yale University, http://www.cs.yale.edu/Linda/linda-lang.html

[Mandlebrot 77] B. B. Mandelbrot, *The Fractal Geometry of Nature*, W.H. Freeman and Company, New York, 1977.

[Manola 99] F. Manola, "Technologies for a Web Object Model", *IEEE Internet Computing*, January/February 1999, pp. 38-47.

[Oreizy 99] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software", *IEEE Intelligent Systems*, May/June 1999, pp. 54-62.

[Peitgen 92] H.-O. Peitgen, H. Jurgens and D. Saupe, *Chaos and Fractals: New Frontiers of Science*, Springer-Verlag, New York, 1990.

[Rucker 95] R. Rucker, Infinity and the Mind: the Science and Philosophy of the Infinite, Princeton University Press, Princeton, New Jersey, 1995.

[Rumbaugh 91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood, NJ, 1991.

[Rumbaugh 98] J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA, 1998.

[Schmidt 96] D. C. Schmidt, M. Fayad and R. Johnson, "Software Patterns", *Communications of the ACM*, October 1996, pp. 37-39.

[Schroeder 91] Schroeder, Manfred. "*Fractals, Chaos, Power Laws*", W.H. Freeman and Company, 1991.

[Srinivasan 99] S. Srinivasan, "Design Patterns in Object-Oriented Frameworks", *IEEE Computer*, February 1999, pp. 24-32.

[Stasko 98] J. Stasko, J. Domingue, M. H. Brown and B. A. Price, *Software Visualization: Programming as a Multimedia Experience*, MIT Press, Cambridge, MA, 1998.

[Voth 98] G. R. Voth, C. Kindel and J. Fujioka, "Distributed Application Development for Three-Tier Architectures: Microsoft Windows DNA", *IEEE Internet Computing*, March/April 1998, pp. 41-45.

[Voyager 99] ObjectSpace Voyager ORB and Universal Container Technology, http://www.objectspace.com/

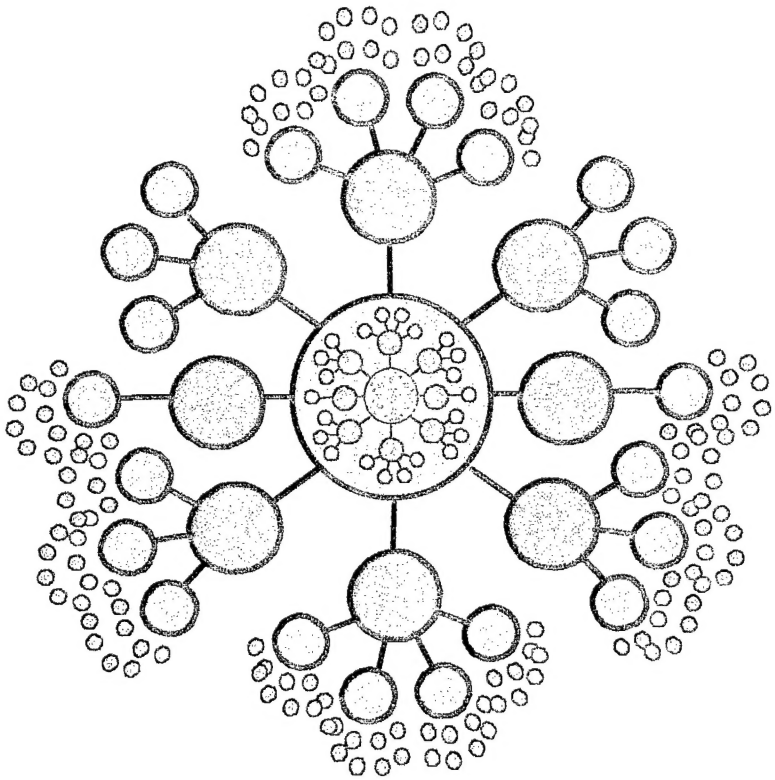
[Waldo 99] J. Waldo, "The Jini Architecture for Network-centric Computing", *Communications of the ACM*, July 1999, pp. 76-82.

[Wooldridge 99] M. J. Woolridge and N. R. Jennings, "Software Engineering with Agents: Pitfalls and Pratfalls", *IEEE Internet Computing*, May/June 1999, pp. 20-27

Open Review Process

During our ongoing efforts into this research we are actively pursuing individual experts in various fields interested in the findings produced thus far. As such, we wish to include these experts in an informal evaluation and open review process to further enhance, validate and stabilize our research findings and ultimate commercialization. Furthermore, it is our desire for this research to not only be successful, but be valuable to commercial and government sectors alike. To increase the probability of value, it is logical to enact an open review process such as this. This model is adopted by many technology forums and consortiums to ensure the appropriate direction and application of technology advances. Some of the organizations where qualified individuals are participating in this open review process include:

- Sun Microsystems
- MITRE
- DISA
- BTG
- GMU (George Mason University)



www.metadapt.com

GNOMON—A form which, when added to some form, results in a new form, similar to the original.